UNIVERSITÀ DEGLI STUDI DI PISA
FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

ESTENSIONE DEL FIREWALL IPFW2 E DEL TRAFFIC SHAPER DUMMYNET AL PROTOCOLLO IPV6

Relatore: Prof. Luigi Rizzo
Correlatori: Prof. Marco Avvenuti

Candidati: Raffaele De Lorenzo, Mariano Tortoriello

ANNO ACCADEMICO 2002-2003

Ai mie genitori
a mia sorella Carmen

Abstract

Lo scopo di questa tesi e' la realizzazione di un firewall con funzionalita' di traffic shaping per sistemi FreeBSD Versione 4.x che sia compatibile con il protocollo IP versione 6 oltre che con il protocollo IP versione 4. Il supporto di entrambi i protocolli è necessario a causa della migrazione graduale degli utenti Internet da IPv4 (protocollo attualmente in uso nella rete) a IPv6 che rappresenta il futuro della rete stessa, nell'ambito di questo aggiornamento i due protocolli possono coesistere, in quanto la progettazione di IPv6 ha esplicitamente previsto il passaggio graduale e la compatibilita', tuttavia la gestione dei due tipi di traffico risulta, per ovvi motivi, differente anche se le necessita' di base di ogni utente non variano e si traducono, appunto, nella richiesta di meccanismi di filtraggio avanzati.

In questa tesi descriviamo tutte le fasi di sviluppo del codice che ci hanno portato alla realizzazione di una versione pienamente funzionante e compatibile con le caratteristiche richieste.

Introduzione

Lo scopo di questa tesi è la realizzazione di un firewall con funzionalitá di traffic shaping per sistemi FreeBSD Versione 4.x che sia compatibile con il protocollo IP versione 6 oltre che con il protocollo IP versione 4[1]. Il supporto di entrambi i protocolli è necessario a causa della migrazione graduale degli utenti Internet da IPv4 (protocollo attualmente in uso nella rete) a IPv6 che rappresenta il futuro della rete stessa, nell'ambito di questo aggiornamento i due protocolli possono coesistere, in quanto la progettazione di IPv6 ha esplicitamente previsto il passaggio graduale e la compatibilitá[2], tuttavia la gestione dei due tipi di traffico risulta, per ovvi motivi, differente anche se le necessitá di base di ogni utente non variano e si traducono, appunto, nella richiesta di meccanismi di filtraggio avanzati.

Nei sistemi FreeBSD versione 4.x l'attuale meccanismo di filtraggio per IPv4 è completo ed affidabile, infatti presenta caratteristiche di traffic shaping curate dal modulo dummynet e interessanti caratteristiche avanzate gestite dal firewall ipfw2 che, come verrá esplicitato in seguito, rappresenta un'evoluzione della vecchia versione ipfw che ha introdotto filtraggi piú potenti ed adatti al traffico di rete attuale. Questi moduli non sono integrati nella gestione dei pacchetti IPv6 e non ne supportano le caratteristiche peculiari.

Per quanto riguarda IPv6, i sistemi FreeBSD Versione 4.x mettono a disposizione un modulo chiamato ip6fw, anche'esso implementato sulla base del vecchio firewall ipfw opportunamente esteso per supportare il nuovo protocollo. Vista la base da cui gli sviluppatori sono partiti, questo firewall non sfrutta le caratteristiche innovative introdotte con la versione 2 e non supporta il traffic shaping pur offrendo un meccanismo elementare di filtraggio.

Viste le caratteristiche del codice dei moduli dummynet ed ipfw2, che permettono una buona espandibilitá a livello operativo, e vista la necessitá di avere un meccanismo di filtraggio in IPv6 con caratteristiche comparabili ad IPv4, si è deciso di estendere dummynet ed ipfw2 al fine di supportare le caratteristiche di IPv6.

Il lavoro necessario per ottenere i nostri scopi si è articolato nella modifica delle seguenti parti di FreeBSD:

- modulo del firewall IPFW2
- modulo del traffic shaper Dummynet

[1] Per semplicitaá in seguito il protocollo IP versione 6 verrá chiamato IPv6, e il protocollo IP versione 4 verrá chiamato IPv4

[2] In [8] e [17] sono ampliamente discusse le problematiche di transizione a livello di host e di router

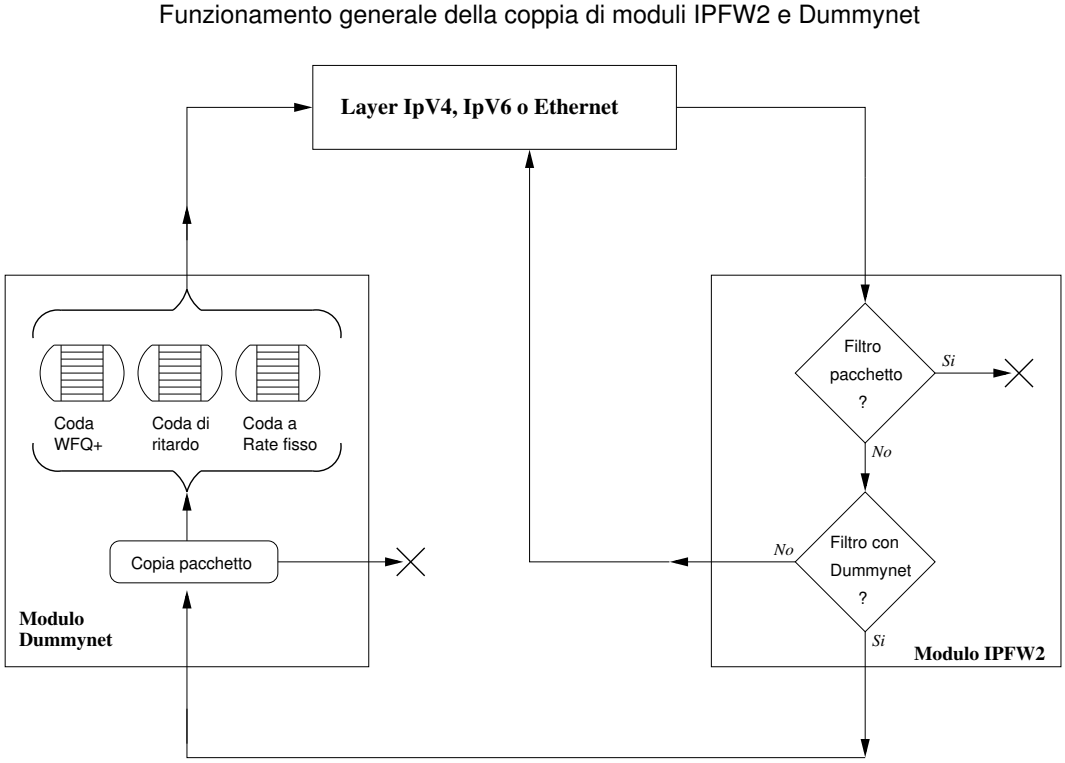

Figura 1: Gestione del pacchetto in presenza dei moduli Dummynet e Ipfw2

- modulo dell'interfaccia con l'utente
- interfacciamento con il layer IPv6

nel firewall vero e proprio abbiamo inserito nuove regole che supportassero il filtraggio di pacchetti IPv6 e nuove funzionalitá di analisi del pacchetto IPv6 che permettessero l'acquisizione dei dati per il filtraggio. In Dummynet abbiamo esteso i meccanismi già funzionanti al supporto del nuovo tipo di pacchetti, introducendo alcune strutture dati aggiuntive e alcune funzioni di interfacciamento. Nell'interfaccia con l'utente abbiamo aggiunto una nuova grammatica di inserimento delle regole IPv6 prendendo spunto dalla precedente versione per IPv4 e mantenendo la compatibilitá con quest'ultima, abbiamo provveduto ad esendere la visualizzazione e il supporto dei nuovi meccanismi di filtraggio. Nel layer IPv6, scritto dal gruppo Kame[3], abbiamo sostituito la chiamata al precedente firewall ed abbiamo inserito gli opportuni agganci in tutti moduli di gestione del protocollo IPv6.

La presentazione del lavoro sopra descritto partirá dalla descrizione dello stato del programma prima del nostro intervento cercando di evidenziare i dettagli che hanno fatto da filoconduttore per la doppia implementazione e, ovviamente, non ci addentreremo troppo nella spiegazione delle scelte che hanno determinato la stesura attuale, ma ci limiteremo ad analizzarne gli aspetti legati al loro aggiornamento. Ad esempio

[3]Uno dei team di sviluppo che curano l'implementazione dello stack protocollare IPv6 e che ha provveduto alla scrittura del codice inserito in FreeBSD, maggiori informazioni sul gruppo sono disponibili al sito ufficiale http://www.kame.net

non tratteremo in maniera esaustiva il meccanismo di traffic shaping, commentando ed analizzando gli algoritmi e, soprattutto, la loro politica di implementazione, ma ne descriveremo le caratteristiche operative generali[4]. Succesivamente illustreremo le differenze che IPv6 ha introdotto, le sue caratteristiche avanzate e cercheremo di esplicitare i punti su cui ci siamo basati per la scrittura del codice. In ultima istanza, descriveremo in maniera dettagliata tutti gli interventi effettuati, cercando di evidenziare i punti concettuali salienti per comprendere il lavoro svolto. In queta parte non tralasceremo la descrizione delle prove effettuate e il loro esito. Per realizzare questa tesi abbiamo utilizzato i classici strumenti di sviluppo presenti nelle distribuzioni standard di FreeBSD, e che sono rappresentati dal compilatore **gcc**, dall'editor **vim** e un programma di analisi del codice chiama **cscope** quest'ultimo ci ha facilitato molto nella comprensione dei moduli da noi modificati sia nella fase di inizio dei lavori sia nella fase di debug.

[4]Un'analisi approfondita sulle politiche e sull'implementazione degli algoritmi di gestione del traffic shaper Dummynet è stata realizzata da Paolo Valente in [1]

Capitolo 1

IPFW2 e Dummynet

1.1 Il firewall IPFW2

L'esigenza di far comunicare piú computer è sempre stata sentita da tutta la comunitá di utilizzatori, tale esigenza si traduce nel supporto delle reti di comunicazione di qualsiasi dimensione. L'aumento delle possibilitá di comunicazione garantito da una tecnologia sempre piú stupefacente e l'estremo sviluppo che in questi anni sta conoscendo la comunitá Internet hanno fatto sviluppare particolarmente anche la necessitá di poter effettuare operazioni di selezione del traffico generato dalla comunicazione al fine di poter controllare i flussi comunicativi. In una rete a commutazione di pacchetto, qual'è Internet, questa esigenza si traduce nell'analisi a qualsiasi livello dell'unitá di trasmissione, cioè del pacchetto. Per rispondere a questa richiesta di controllo sono stati sviluppati programmi che consentono di applicare politiche di filtraggio sempre piú avanzate, tali programmi prendono il nome di firewall.

In generale, quindi, un firewall è un meccanismo di analisi dei pacchetti che giungono al sistema dalle interfaccie di rete, il firewall si inserisce tra la gestione operativa del pacchetto (frammentazione, analisi di correttezza ecc.) e l'elaborazione dello stesso, sia essa rivolta alla risposta automatica dei messaggi di rete (risposte ICMP ecc.) oppure inoltrata alla gestione dei protocolli di livello superiore.

La gestione e il controllo del traffico di rete si sposta quindi sulla politica di configurazione del firewall e sulle caratteristiche di filtraggio che esso offre. Nello standard IPv4 che sta alla base della comunicazione della comunitá Internet si sono sperimentate e risolte necessitá di controllo e filtraggio molto variegate visto il numero rilevante di computer messi in comunicazione da Internet

1.1.1 Cos'è il firewall IPFW2

Il programma IPFW2 consente il filtraggio dei pacchetti Internet e viene utilizzato dal sistema operativo FreeBSD a partire dall'estate del 2002, IPFW2 è un arricchimento della vecchia versione del firewall di FreeBSD che si chiamava IPFW. Come recita il manuale di FreeBSD [2], con la versione 2 del firewall sono state introdotte parecchie migliorie dal punto di vista delle funzionalità offerte, prima di passare al dettaglio implementativo, ne elenchiamo i punti salienti:

- Possibilità di gestire pacchetti anche non IPv4

- Specificazione delle porte nel filtraggio di pacchetti a livello TCP e UDP
- Possibilità di organizzare l'insieme di regole in gruppi
- Possibilità di filtraggio a livello MAC
- Regole stateful
- Blocchi OR nelle regole

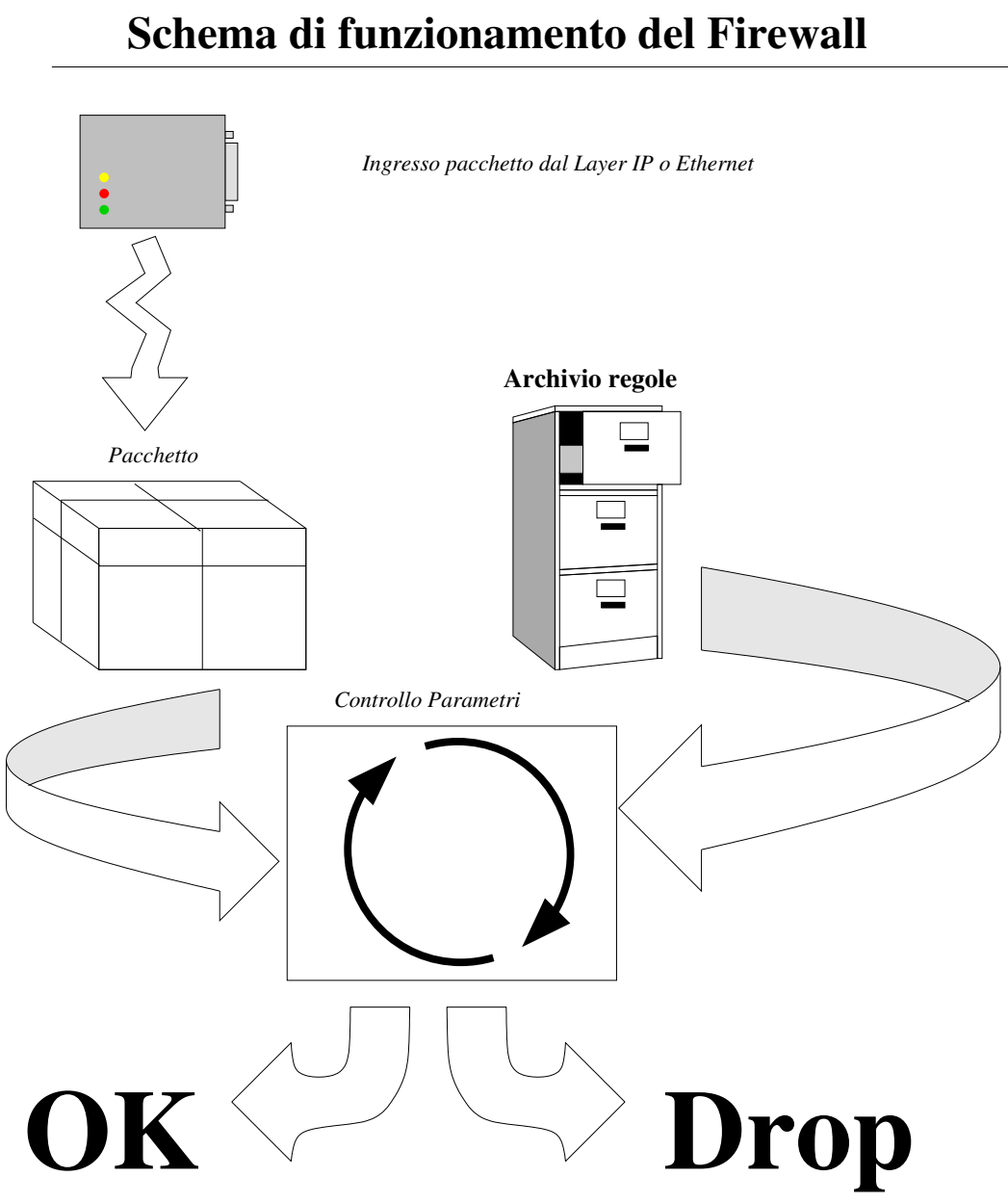

Figura 1.1: Schema di funzionamento di ipfw2

Queste funzionalità rendono il firewall altamente configurabile ed adatto pienamente a qualsiasi politica di limitazione di traffico o protezione da attacchi. E' bene

specificare che il firewall in se non garantisce l'inattaccabilità del sistema, ma permette al gestore del sistema stesso di programmare un insieme di regole che premettono di raggiungere questo obiettivo. La potenza di questo firewall sta, appunto, nella possibilitá di porre filtri su ogni parte del pacchetto IP e di alcuni parametri dei protocolli di livello superiore. Con questa caratteristica fondamentale, la sicurezza e il controllo si applica con una programmazione delle regole e sull'impostazione della loro scansione

1.1.2 Funzionamento del firewall

Il modulo ipfw2 è composto da due parti distinte, una parte operativa e una parte di controllo. La parte di controllo dialoga con l'utente attraverso l'interfaccia utente che gestisce l'operatività del firewall intesa come impostazioni di funzionamento e gestione delle regole. Un'altra via di impostazione delle modalità operative del firewall avviene attraverso l'impostazione di alcune variabili del kernel (variabili *sysctl*[1]), quest'ultima via peró serve unicamente ad impostare il comportamento generale del firewall e non le regole, si veda [2] per i dettagli. Tramite questi canali l'utente (che coincide con l'amministratore del sistema) stabilisce la reazione del sistema ai tentativi di comunicazione con e dall'esterno.

La parte operativa è il cuore vero e proprio del firewall ed è la routine che viene invocata per l'analisi delle regole. Questa analisi avviene attraverso una scansione sequenziale delle regole, le quali sono numerate in serie crescente e vengono verificate sequenzialmente dal firewall prendendo come riferimento il loro numero d'ordine dalla prima sino all'ultima regola che è detta *regola di default* . La prima regola che è compatibile con il pacchetto arrivato viene applicata. La sequenza di inserimento (sia essa automatica tramite script, o manuale tramite interfaccia di testo) determina, in mancanza di specifica indicazione, un'attribuzione automatica e crescente del numero d'ordine. Anche quest'ultimo aspetto può essere governato attraverso la parte controllo.

La regola di default è l'ultima e cioé la numero 65535, ed è speciale in quanto non puó in essere modificata tramite interfaccia utente. La sua impostazione viene decisa all'atto della compilazione del modulo attraverso la presenza o meno della clausula IPFIREWALL_DEFAULT_TO_ACCEPT nel *Makefile*[2], questa clausula causerà l'inserimento di una regola mirata a corrispondere ad ogni pacchetto e il cui *match*[3] è sempre verificato. La semantica, come suggerisce il nome, è la seguente: se la direttiva è presente la regola di default permetterà il passaggio di ogni pacchetto, viceversa ogni pacchetto verrà scartato.

Cerchiamo ora di capire quando viene invocata l'analisi del firewall, a questo proposito facciamo riferimento alla figura 1.2 presente in [2]. La chiamata al firewall avviene quindi in svariati punti dello stack protocollare per cui un solito pacchetto può

[1] Comando di impostazione delle variabili di kernel, si faccia riferimento alla pagina del manuale corrispondente visualizzabile tramite shell digitando il comando *man sysctl* oppure su internet all'indirizzo *http://www.FreeBSD.org/cgi/man.cgi?query=sysctl*

[2] File contenente le regole di compilazione del modulo, per il modulo ipfw questo file si trova nella directory */usr/src/sys/modules/ipfw*

[3] Corrispondenza tra i parametri della regola e i parametri del pacchetto

invocare più chiamate del firewall a seconda delle decisioni dell'amministratore del sistema.

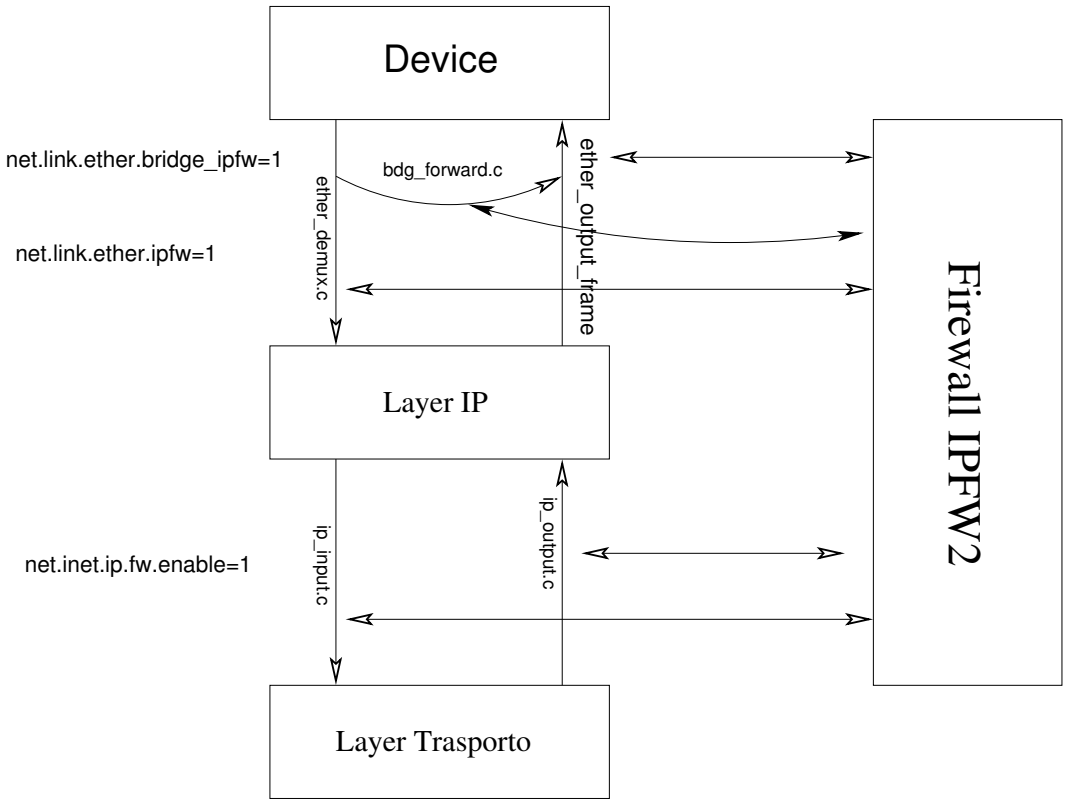

Figura 1.2: Schema di chiamata del firewall ipfw2 per lo stack IPv4

In particolare evidenziamo dal chiamata dal layer ethernet e la chiamata dal layer IP, si nota come le chiamate avvengano nelle due direzioni di ingresso e di uscita. Questo sottolinea come IPFW2 possa essere usato non solo per prevenire attacchi dall'esterno ma anche per limitare le richieste di traffico all'esterno, e quindi vincolare l'interno ad una comunicazione controllata.

1.1.3 Struttura delle regole

Le regole sono organizzate secondo una lista di *struct ip_fw* [4]. Questa struttura contiene, oltre ad alcune informazioni statistiche (contatore di pacchetti che sono risultati conformi alla regola, byte transitati nei pacchetti relativi e timestap dell'ultimo pacchetto conforme) e ad alcuni campi per velocizzare l'elaborazione (puntatore alla regola skipto), il corpo della regola stessa. Il corpo della regola contiene la codifica delle istruzioni di filtraggio, in IPFW2 ogni regola è composta da più microistruzioni ognuna relativa ad una particolare parte del pacchetto IP o di livello superiore nonché l'azione che deve essere intrapresa per la sua gestione.

La base di queste microistruzioni è la struttura *ipfw_insn*. Quest'ultima contiene un codice che identifica il tipo di istruzione, la sua dimensione e un piccolo parametro aggiuntivo per eventuali informazioni ausiliarie. Le dimensioni ridotte di questa struttura non precludono la presenza di parametri complessi di dimensione maggiore, tali parametri dovranno peró essere accodati alla struttura base e la loro dimensione dovrá essere conteggiata nel campo apposito della stessa. Per alcuni comandi di filtraggio di uso comune che necessatano di strutture dati particolari sono stati definiti tipo particolari che facilitano la stesura del codice relativo.

La struttura della regola è composta da due parti

- azione
- comando di filtraggio

Entrambe le parti constano di una o più strutture *ipfw_insn* seguite da opportuni parametri secondo le necessitá operative derivate dal comando stesso. I comandi supportati sono elencati nella *enum ipfw_opcodes* e comprendono entrambe le classi sopra esposte. Un comando di filtraggio, quindi, comprende una serie di microistruzioni, ognuna di esse è un piccolo blocco che il firewall analizza per determinare se il pacchetto corrisponde o meno alla regola in analisi.

Le microistruzioni possono discriminare ogni parte del pacchetto IP analizzato, e vanno dai contenuti dell'header (ad esempio gli indirizzi sorgente e destinazione) fino al contenuto del pacchetto IP e quindi all'header del protocollo di livello superiore (ad es. le porte del protocollo UDP/TCP). Come si è detto poc'anzi, per comoditaá alcuni comandi di utilizzo frequente hanno un loro tipo dedicato al fine di semplificare la stesura del codice, ad esempio *ipfw_insn_sa* è la struttura destinata a contenere un socket IPv4 mentre *ipfw_insn_ip* è stata creata per contenere un indirizzo IPv4 con la relativa maschera di rete, queste definizioni di comodo rispecchiano la struttura sopra descritta: hanno, infatti, al primo posto una struttura *ipfw_insn* seguito da un opportuno tipo (nei due esempi specificati sopra una *struct sockaddr_in* e due *struct in_addr*).

L'azione descrive quello che il firewall deve fare una volta stabilito che il pacchetto in analisi è conforme alla regola rappresentata dai vari comandi di filtraggio sopra descritti. Le azioni tipiche sono il rifiuto del pacchetto (DENY), l'accattazione del pacchetto (ACCEPT). Esistono anche azioni diverse come il passaggio a dummynet (*O_PIPE* e *O_QUEUE*), nonchè istruzioni di salto ad altre regole. Questo tipo di microistruzioni hanno la stessa struttura dei comandi di filtraggio con la differenza che,

[4] I tipi di dato citati in queste note fanno riferimento al file *ip_fw2.h*.

dovendo codificare uno spettro di possibilitá non molto elevato, si riesce a gestire completamente l'azione con il semplice utilizzo di una *ipfw_insn* per azione. Questo ovviamente non preclude assolutamente alla creazione di azioni con complessitá maggiore e che necessitino di parametri piú complessi.

La prima struttura *ipfw_insn cmd* presente nella regola, rappresenta il primo comando di filtraggio della regola stessa, eventuali comandi di filtraggio successivi vengono allocati in maniera contigua e la loro dimensione è stabilita da un apposito parametro della struttura stessa che il modulo di creazione della regola dovrà impostare. Sempre contiguamente vengono allocate anche le azioni, l'offset dell'inizio delle azioni è a sua volta disponibile come parametro all'interno della struttura base della regola al fine di permettere la loro individuazione rapida.

Da questa descrizione dovrebbe essere chiaro l'ulteriore punto di forza di questo firewall che è rappresentato dalla facilitá di espansione delle possibilitá di filtraggio, infatti la creazione di una nuova regola é concettualmente molto semplice e consiste nell'aggiungere un nuovo *opcode* (con eventualmente una struttura dedicata) e la sua gestione completa sia a livello di riempimento delle opzioni relative sia a livello di analisi di conformitá.

1.1.4 Regole statiche e regole dinamiche

Una regola scritta per ipfw2 può essere di tipo statico e di tipo dinamico, la differenza che intercorre tra queste due categorie è rappresentata dalla loro gestione operativa.

Si considerano regole statiche quell'insieme di regole la cui gestione viene effettuata tramite l'analisi del solo pacchetto che giunge al firewall e il loro comportamento è univocamente stabilito dalla regola stessa e non viene alterato dal flusso di traffico che il firewall sperimenta. A questo insieme appartengono tutte quelle regole di filtraggio standard che interessano l'header del pacchetto di livello IP e di livello TCP/UDP, e si limitano all'osservazione e filtraggio dei solo parametri presenti negli header stessi.

Una regola dinamica, invece, è una regola la cui gestione determina la creazione di strutture dati temporanee interne al firewall che servono a distinguere ed aggiornare il comportamento operativo del firewall rispetto al traffico che si osserva. A questo tipo di regole appartengono ad esempio quei comandi di filtraggio che limitano le connessione TCP/UDP in ingresso/uscita.

1.1.5 Come è implementato

Nella sezioni precedenti abbiamo accennato alla divisione logica del funzionamento del firewall tra parte operativa e parte controllo. In questa sezione cercheremo di approfondire questi aspetti con alcuni dettagli implementativi ulteriori. Anzitutto precisiamo che tale divisione logica corrisponde, a livello implementativo, con la presenza di due punti di aggancio distinti che il modulo esporta a tutti gli utilizzatori.

La parte controllo

La parte controllo si invoca aprendo un socket comunicativo opportunamente nominato, tale canale comunicativo viene gestito nel modulo attraverso la chiamata al puntatore *ip_fw_ctl_ptr* il quale fa riferimento alla funzione *ipfw_ctl* che si occupa della gestione delle regole intesa come organizzazione della lista. Tramite questa interfaccia è possibile:

- Ottenere l'elenco delle regole disponibili
- Aggiungere/cancellare regole
- Raggruppare regole in set
- Azzerare o visualizzare i contatori relativi ad ogni regola

La parte controllo, quindi, si assume l'onere di modificare la lista delle regole che sono presenti nel modulo. Questo compito si traduce in opportune chiamate alle funzioni deputate alle gestione della lista di regole, le quali, soprattutto per quanto riguarda l'inserimento, effettuano alcuni semplici controlli di coerenza della regola che, nel caso tipico, si limitano a controlli sulle dimensioni delle microistruzioni ricevute. La comunicazione tra spazio utente (interfaccia grafica) e spazio kernel (modulo ipfw) avviene tramite un socket che la parte controllo gestisce.

La parte operativa

La parte operativa viene invocata attraverso il puntatore *ip_fw_chk_ptr* il quale fa riferimento alla funzione *ipfw_chk*. Questa funzione si occupa delle operazioni di filtraggio vere e proprie. I *punti di aggancio* dove avviene la chiamata al firewall si trovano nei moduli di gestione delle comunicazioni ethernet *if_ethersubr.c* e *bridge.c*[5], nonchè nei moduli di gestione del protocollo IPv4 *ip_input.c* e *ip_output.c*[6] come si vede dalla figura 1.2.

La parte operativa viene invocata con alcuni parametri tutti memorizzati in una struttura caratteristica chiamata *args*, all'interno di questa struttura (il cui utilizzo è condiviso con dummynet) il firewall riceve alcuni parametri operativi, quali il pacchetto sul quale effettuerá le proprie operazioni di filtraggio, alcuni flag operativi e un puntatore all'ultima regola analizzata qualora il firewall fosse giá stato invocato precedentemente. La parte operativa, non appena invocata, inizia a collezionare localmente tutte le informazioni di filtraggio che è possibile ricavare dal pacchetto ricevuto al fine di velocizzare l'operazione di analisi delle regole prelevando pure alcune informazioni generali utili al modulo dummynet qualora fosse invocato. Dopo aver reperito le informazioni di base il firewall inizia ad analizzare sequenzialmente tutti i comandi di tutte le regole cercando di confrontare tutti i campi che localmente sono disponibili rispetto alla richiesta della regola. La conformità[7] del pacchetto ad una regola è stabilita se

[5] questi file di sistema si trovano nella cartella /usr/src/sys/net

[6] questi file di sistema si trovano nella cartella /usr/src/sys/netinet

[7] Conformitá, corrispondenza o *match* sono per noi sinonimi. Queste dizioni si mescoleranno nel proseguire della trattazione

i parametri operativi estratti dall'header e dal corpo corrispondono a tutti i comandi di filtraggio presenti nella regola, cioè se tutte le microistruzioni che compongono la regole sono verificate. Se la corrispondenza è esatta viene eseguita l'azione relativa.

Il pacchetto, per essere accettato, deve corrispondere obbligatoriamente ad una regola dell'insieme totale in caso contrario viene scartato e segnalato un errore, questo comportamento giustifica la presenza della regola di default. Questa regola, che dovrebbe essere la regola piú generale che effettua il match con ogni pacchetto, permette all'utente di decidere il comportamento del firewall nel caso generale. La presenza della regola precedentemente analizzata nella lista dei parametri, fa capire al modulo IPFW2 che la sua invocazione sul pacchetto é giá avvenuta, il suo comportamento a questo punto puó essere deciso sulla base di un'apposita variabile operativa, impostabile sia tramite una variabile *sysctl*[8] oppure tramite una chiamata all'interfaccia di comando. In particolare il firewall può riprendere l'analisi delle regole dall'ultima analizzata in poi oppure può ritenere la regola su dummynet come l'ultima da analizzarsi e quindi terminare senza proseguire.

[8]Il nome della variabile è *net.inet.ipfw.one_pass*

1.2 Dummynet: il traffic shaper

1.2.1 Descrizione generale

Dummynet è il modulo che permette di modellare il traffico IP che viaggia tra le varie interfacce di rete emulando dei canali di trasmissione configurabili. Tramite l'interfaccia standard di ipfw2 è possibile, infatti, configurare come modellare il traffico attraverso le politiche disponibili. Il principio di funzionamento è molto semplice e puó essere spiegato attraverso un'analogia idraulica: ogni regola di traffico puó essere vista come un rubinetto la cui politica di apertura è decisa dalla regola stessa, ed il flusso di acqua uscente dal rubinetto rappresenta la banda assegnata. Dummynet permette diversi tipi di regolazione del flusso di dati. A seconda del tipo di coda e dei parametri caratteristici viene deciso il comportamento per il traffico. Le code possono essere di tre tipi differenti

- Code a rate fisso
- Code di ritardo (delay)
- Code WF2Q+

Le code a rate fisso permettono il settaggio della banda in maniera permanente, nel senso che una volta settata, la banda rimane tale finché il root non modifica manualmente la regola. Le code di ritardo servono esclusivamente per ritardare la gestione del pacchetto da parte dell'applicazione cui si riferisce. Queste due tipi vengono identificati con il termine *pipe*, e vengono gestite esattamente con l'analogia idraulica suggeriscce e cioé finché il traffico è conforme alle specifiche dettate dalla configurazione della pipe viene approvato e quindi non ostacolato, viceversa, viene fermato. Le code WF2Q+ servono per stabilire all'interno di un macro flusso, diversi flussi di dati che accedono alla risorsa secondo un peso definito al momento della creazione. Minore è il peso maggiore sará la garanzia di banda che avrá la coda, il tutto senza privare la coda con peso minore di una garanzia minima di banda che permetta una quota C'è da precisare un particolare, in quanto è vero che i tipi di code sono tre ma in generale le code a rate fisso e quelle WF2Q+ hanno sempre in uscita un modoulo di ritardo, quindi in pratica è come se fossero una coppia di code poste in cascata, in cui il secondo stadio è sempre una coda di ritardo. Per chiarire meglio tutti questi concetti basta osservare lo schema riassuntivo che è visualizzabile nella figura 1.3.

Dummynet utilizza la stessa interfaccia utente di IPFW2 per l'inserimento delle regole, il flusso del pacchetto diviene il seguente in presenza di una regola settata

Come anticipato nell'indtroduzione, non ci addentreremo nella descrizione degli algoritmi[9] ma vedremo piú che altro come un utente vede la configurazione di dummynet e come debba essere utilizzato. I riferimenti piú significativi rimangono [2] e [3].

[9] A proposito si consideri [1]

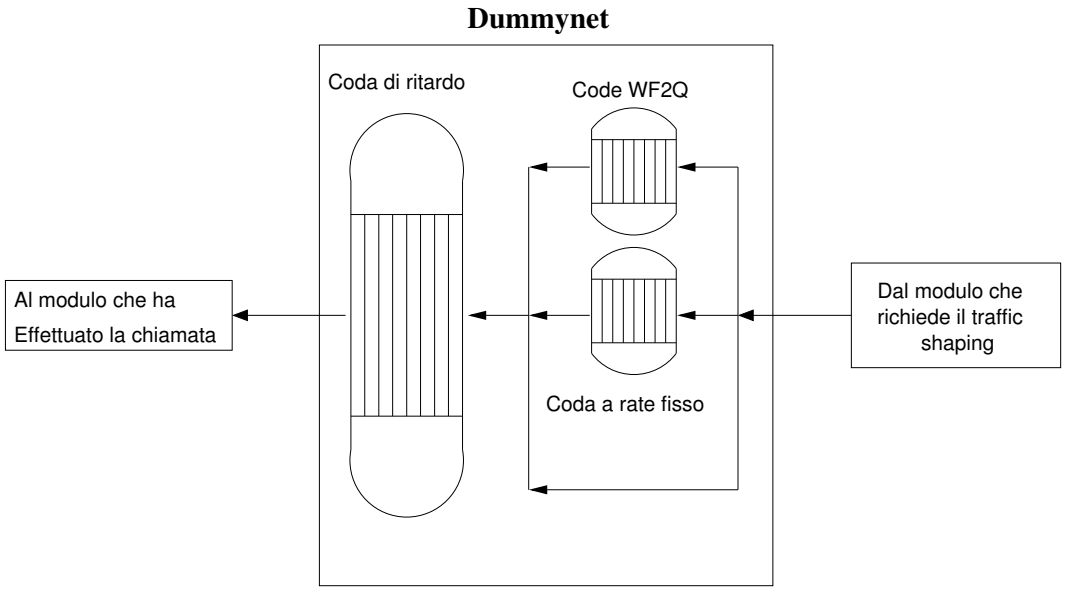

Figura 1.3: Funzionamento generale di Dummynet

1.2.2 Implementazione

Cosí come ipfw2 anche l'implementazione di dummynet è scissa in piúparti ognuna con il compito di gestire un compito specifico. In particolare per dummynet si possono inviduare tre nuclei operativi che possono essere definiti:

- parte controllo
- parte operativa
- gestione interfacciamento

La parte controllo di dummynet

La parte controllo ha scopi del tutto simili a quelli descritti per il firewall ipfw2 e si occupa quindi di ricevere ed impostare le configurazioni e gli ordini di gestione delle *pipe/queue* dall'interfaccia di configurazione. La funzione che si occupa di questa gestione si chiama *ip_dn_ctl*.

Vediamo a questo punto come avviene la configurazione di una *pipe* o di una *queue*. Una prima configurazione del flusso di dati che dovrá passare per il traffic shaper viene effettuata attraverso la regola di filtraggio e di aggancio nel firewall. La regola in questione si occuperá di selezionare il traffico sulla base delle informazioni statiche presenti nel pacchetto quali indirizzo sorgente e destinazione, tipo di pacchetto del livello di trasporto, porta di ingresso/uscita ecc. Successivamente si applicheranno le politiche di gestione del traffico selezionato che, data la differente natura dei due tipi

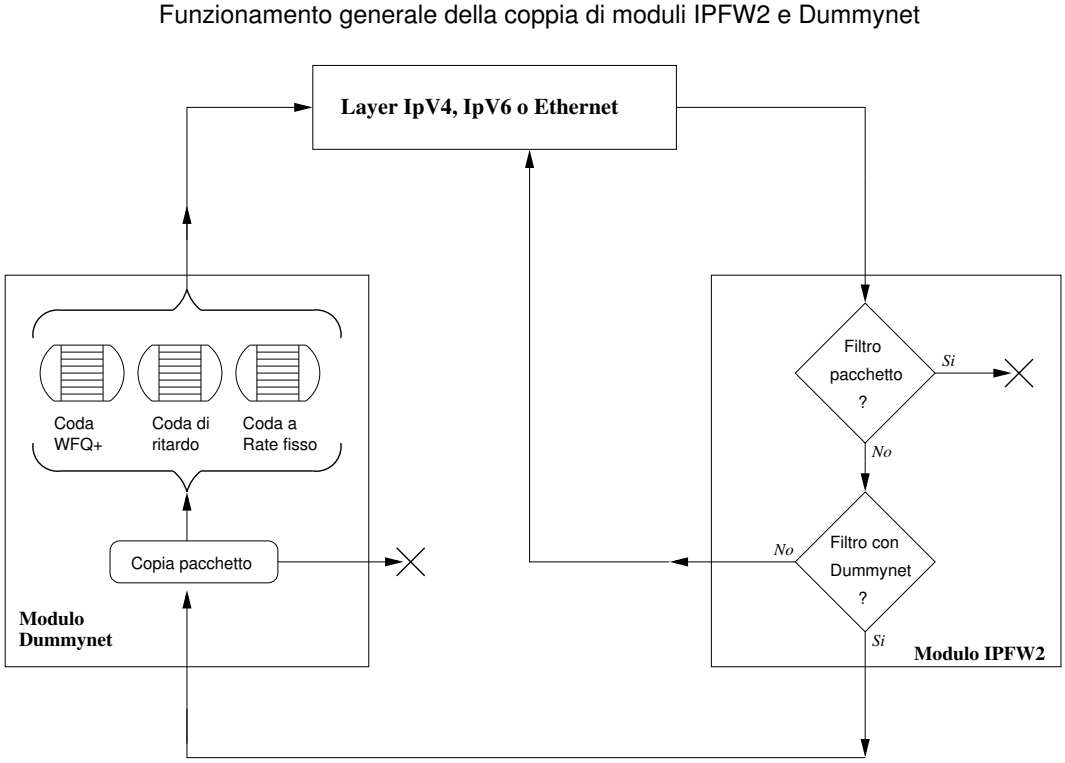

Figura 1.4: Gestione del pacchetto in presenza dei moduli Dummynet e Ipfw2

di flusso, genereranno una configurazione differente anche se alcuni parametri posso essere usati in entrambe le configurazioni[10].

Le configurazioni comuni ad entrambi i flussi di dati piú rilevanti sono rappresentate della seguenti opzioni:

- **mask**: serve effettuare ulteriori cernite sul flusso di dati da trattare attraverso una maschera di identificazione che può comprendere gli indirizzi sorgente e destinazione oppure le porte del protocollo di livello di trasporto
- **plr**: é il packet loss ratio e stabilisce la probabilitá con cui il pacchetto viene considerato perso. Questo parametro é molto utile nel test di programmi che sfruttano la rete.
- **red/gred**: algoritmi di gestione del traffico

Per quanto riguarda il primo tipo di flusso e cioé una *pipe*, la richiesta di parametri si esaurisce con l'inserimento statico di una serie di parametri che caratterizzano il traffico che sono:

- **bw**: che sta per *bandwidth* e rappresenta l'apertura di banda che si desidera per il canale
- **delay**: rapprensenta il ritardo che si vuole imporre al traffico selezionato

[10]A questo proposito si veda [2] nella sezione relativa alla configurazione di dummynet, in questa sede analizzeremo solo le configurazioni piú generali e quelle che hanno influenzato il nostro lavoro

La configurazione di una *queue*, invece, ha come parametri caratteristici i seguenti:

- **pipe**: la pipe cui si puó far corrispondere il flusso di traffico per ulteriori filtraggi
- **weight**: peso da attribuirsi alla queue per l'applicazione dell'algoritmo di *fair queuing*

Queste configurazioni vengono memorizzate nelle apposite strutture *dn_pipe*.

Interfacciamento con i layer di gestione del pacchetto

La gestione interfacciamento si occupa di prelevare il pacchetto dal chiamante,di memorizzare le informazioni relative al flusso di dati ed, infine, di inserire il pacchetto prelevato nelle code di gestione. La realizzazione delle politiche di modellazione del traffico é realizzata tramite la copia del pacchetto all'interno di code speciali proprie del modulo, la funzione che si occupa di gestire questo compito si chiama *dummynet_io*. Dal punto di vista operativo, quando viene invocato Dummynet viene effettuata, ad opera di questa funzione, una copia del pacchetto creando una struttura *dn_pkt*, tale struttura ha la forma di un *M_TAG* e si occupa di contenere come elemento successivo della catena *mbuf* il pacchetto appena prelevato dallo stack protocollare nonchè di alcuni parametri utili alle operazioni successive presenti nella struttura *args* che rappresenta la struttura dedicata al passaggio del parametri per firewall e dummynet[11]. I dati presenti in questa struttura sono in gran parte prelevati dal firewall IPFW2, che si occupa di memorizzarli in appositi campi della struttura *args*, e vengono utilizzati per identificare il flusso di dati. Tali dati caratteristici del pacchetto, quali indirizzo di destinazione e sorgente, porte TCP/UDP interessate, identificano il flusso di dati all'interno della pipe/queue che è stata riconosciuta dalla precedente chiamata al firewall. Nella figura 1.4 si vede appunto che la chiamata al firewall é precedente a quella di dummynet, in particolare il firewall in presenza di una apposita regola con azione *O_PIPE* oppure *O_QUEUE* capisce che deve essere invocato dummynet e ritorna al chiamante riportando il parametro dell'azione corrispondente che non è altro che il numero di pipe/queue cui dummynet dovrá far riferimento. Dopo essere stato invocato dummynet provvede a creare un flusso all'interno della pipe/queue (sempre che sia possibile crearne piú di uno) attraverso i parametri identificativi del pacchetto che IPFW2 ha provveduto a memorizzare, dopo di che inserisce il tutto nella coda corrispondente e fa espirare la gestione relativa del pacchetto.

Visto il tipo di gestione, che determina la morte di una copia del pacchetto passato al traffic shaper, si rende necessario effettuare una copia fisica del route calcolato per il pacchetto e l'interfaccia cui il pacchetto é destinato. Nel caso di flusso di ingresso questa memorizzazione non viene effettuata in quando non ancora completata dal layer di gestione e d'altra parte inutile cisto che la gestione effettiva del pacchetto deve ancora essere eseguita. In fase di uscita, invece, questa necessitá risulta indispensabile per il buon funzionamento del modulo. Per questo motivo il chiamante di dummynet effettua una copia dei parametri operativi di routing del pacchetto che, in ultima analisi, altro non sono che i campi *struct route* e *struct ifnet* * che rapprensentano, in ordine, i

[11] Questo tipo di struttura é stata abolita nella *-current*, il porting a queste modifiche è in atto ed in fase di testing

parametri di routing del pacchetto e l'interfaccia cui il pacchetto dovrá essere inoltrato. Da notare la differenza che esiste tra i due parametri il primo viene copiato per intero, in quanto struttura dinamica all'interno del sistema che viene eliminata quando la gestione del pacchetto relativa termina e che quindi nel nostro caso deve essere salvata come copia, del secondo, invece, si memorizza il solo puntatore poiché esso rappresenta un interfaccia di sistema che dovrebbe permanere durante l'intera accensione del sistema.

La parte operativa di dummynet

La parte operativa del modulo dummynet si occupa della realizzazione di quanto viene preparato dalle altre subroutine di gestione. Ad ogni quanto temporale, impostabile attraverso un opzione di compilazione[12], viene eseguita la routine *dummynet* che si occupa di esaminare tutte le code del traffic shaper ed intraprende le dovute azioni riguardo la gestione dei pacchetti presenti nelle code stesse. All'interno di ogni struttura dati della coda, oltre ai dati identificativi del flusso di traffico, sono presenti le impostazioni delle politiche con cui deve essere gestito il pacchetto e quindi eventuali *time line*. La routine *dummynet* provvedere quindi a inoltrare i pacchetti eventualmenti pronti per la trasmissione verso l'interfaccia da cui sono stai prelevati e che la funzione di archiviazione *dummynet_io* ha provveduto a memorizzare nell'apposito campo *dn_dir* della struttura di memorizzazione del pacchetto *dn_pkt*.

Le azioni che questa routine puó effettuare sono l'aggiornamento dei parametri della struttura che contiene il pacchetto conformemente all'algoritmo configurato, l'inoltro del pacchetto al layer relativo quando la gestione dello stesso é esaurita sempre conformemente alle politiche impostate.

Dopo aver preso la decisione di inoltrare il pacchetto, il controllo ripassa ai layer di gestione che identifica il pacchetto come proveniente da dummynet attraverso un TAG opportuno, come si è esaminato prima, e provvedono ad informare il firewall circa il precedente match che il pacchetto ha subito. Il comportamento del firewall a questo punto dipende dall'impostazione della variabile *sysctl one_pass* tramite la quale sceglie se continuare l'analisi del set di regola da quella che ha delegato la gestione del pacchetto a dummynet, oppure ritenere il filtraggio esaurito. Entrambe le scelte hanno senso a patto che il set di regole sia coerente con la scelta stessa, l'applicazione di questi metodi sposta, come si é detto, il concetto di sicurezza dal hardware alla logica di programmazione delle regole.

[12] Si puó impostare a livello di opzione di compilazione del kernel la variabile *HZ* che determina appunto il quanto temporale di esecuzione della routine di gestione delle code

1.3 Interfaccia con l'utente

1.3.1 Descrizione generale

Esiste un'unica interfaccia che permette l'interazione tra l'utente ed i due moduli Dummynet ed IPFW2, tale programma prende il nome di ipfw. Prendendo come riferimento il solito manuale di sistema [2], si nota che, al di lá delle opzioni modali che precedono il comando vero e proprio, in generale la prima opzione che compare rappresenta il comando che determina l'operazione che l'interfaccia grafica si accinge ad effettuare. I comandi possibili sono:

- *add*/*delete*: per impostare le regole al firewall
- *list* o *show*: per visualizzare le regole attive
- *enable*/*disable*: per governare le modalitá operative
- *pipe*/*queue*: per impostare dummynet

Di seguito procederemo nell'analisi delle varie operazioni e ne daremo una descrizione implementativa di massima, questo allo scopo di rendere piú comprensibili le descrizioni delle modifiche che si sono rese necessarie per l'aggiornamento al protocollo IP versione 6.

L'opzione add, come suggerisce la traduzione, serve ad aggiungere regole al firewall o pipe/queue a Dummynet. La sintassi del comando, tralasciando le opzioni piú dettagliate, è in generale la seguente:

ipfw ***add*** *[corpo della regola]*

il corpo della regola a sua volta ha una struttura abbastanza semplice:

azione protocollo ***from*** *sorgente* ***to*** *destinazione [opzioni]*

Le azioni che supportano questa sintassi sono:

- allow/permit/accept/pass
- deny/drop

e sono le azioni che si riferiscono al filtraggio dei pacchetti nei protocolli supportati che devono essere riportati subito dopo l'azione. La struttura della regola a questo punto prevede la La regola poi prevede l'indicazione degli indirizzi relativi alla sorgente alla destinazione: la notazione prevista per IPv4 è la classica decimale puntata, inoltre è possibile indicare una *subnet mask* ed anche definire un insieme di indirizzi. Tali possibilitá di aggregazione, rendono particolarmente configurabili le regole.

Per esempio, supponiamo di volere inserire una regola per bloccare tutti i pacchetti provenienti dall'indirizzo IPv4 192.168.0.2, allora da linea di comando si scriverá:

ipfw add deny ip from 192.168.0.1 to 192.168.0.20

dove la *add* sta a significare l'aggiunta di una nuova regola, *ip* è il protocollo, *from indirizzo* indica l'indirizzo sorgente, mentre *to indirizzo* indica il destinatario.

Quella appena descritta è la sintassi base di un comando di filtraggio minimo, oltre a questa serie di comandi base, si possono inserire una serie molto nutrita di opzioni che possono interessare ogni campo dell'header IPv4 (escluso il checksum, come ovvio) e anche dell'header del protocollo di livello superiore di tipo TCP o UDP. La sintassi dettagliata di queste opzioni dipende dalla natura del campo che sono destinati a filtrare, in questa sede ci sembra opportuno sottolineare la possibilitá di inserire un filtraggio sulle porteTCP/UDP attraverso il comando *src-port* e *dst-port*.

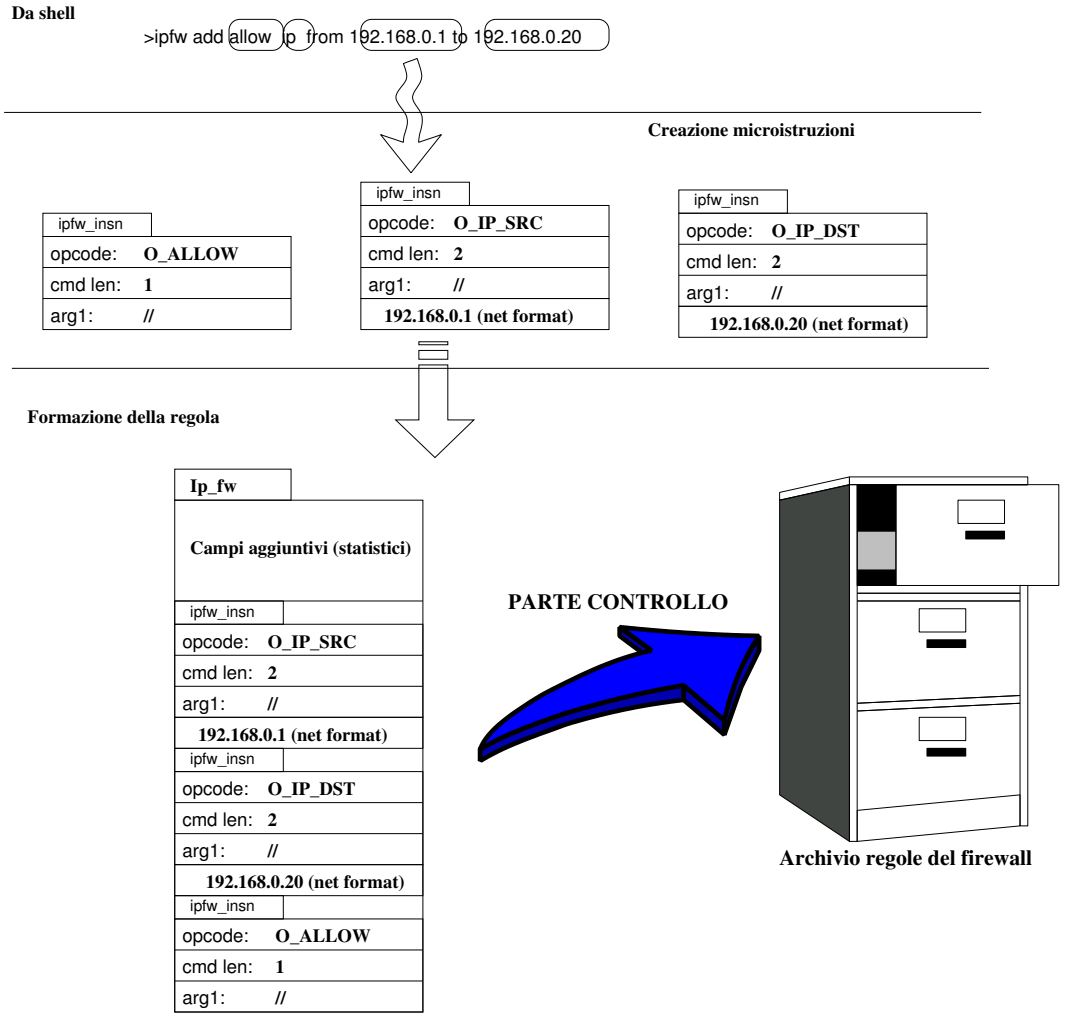

Figura 1.5: Esempio di traduzione di una regola IPv4

Oltre ai comandi veri e propri di filtraggio, la cui sintassi è stata appena descritta, ci sono una serie di comandi operativi che impostano il comportamento del firewall durante l'analisi dell'insieme di regole. Questi comandi permetto di modificare la sequenza delle regole, di inserire regole speciali che permettano salti all'interno del flusso, di impostare insiemi di regole ecc., inoltre ci sono alcuni comandi operativi che permettono di visualizzare le regole attive, impostare alcuni parametri operativi (debug, logging ecc.) nonchè di resettare i contatori statistici che il firewall incorpora nella regola. Fanno parte di questa serie di categorie comandi del tipo:

- **ipfw enable debug**
- **ipfw show**
- **ipfw pipe show**
- **ipfw flush**

con questi comandi si è ordinato al firewall quanto segue (secondo l'ordine di scrittura) di attivare la modalitá di debugging, di mostrare l'insieme delle regole attive con relativi contatori, di mostrare i flussi di traffico del traffic shaper con i relativi contatori, di azzerare completamente l'insieme delle regole. Chiaramente in questa sede abbiamo mostrato esempi, la sintassi completa risulta molto piú articolata e completa.

Per quanto riguarda il traffic shaper Dummynet la sintassi è assolutamente analoga all'inserimento di una regola standard. Infatti, prima poter essere inviato al traffic shaper vero e proprio il pacchetto ha bisogno di soddisfare alcuni parametri caratteristici che possano identificare il flusso di traffico, questi parametri sono, come minimo, gli indirizzi sorgente e destinazione, ma é possibile impostare un filtraggio di traffico molto fine che possa ricoprire (come nel caso precedente) tutti i parametri del pacchetto.

Una regola di questo tipo sostituirá il comando che determina l'azione con il comando *pipe* se si vuole impostare una un traffico a rate fisso o con ritardi oppure il comando *queue* se si vorrá impostare un traffico secondo la politica WF2Q+[13]. Solo dopo questa prima identificazione il pacchetto viene inoltrato al traffic shaper vero e proprio che provvederà ad applicare la politica che é stata impostata per quel flusso di traffico che a quel punto è univocamente determinato. Facendo un esempio se volessimo inserire una limitazione di banda pari a 30Kbit/s a tutti i computer in rete, sia in ingresso che in uscita:

- **ipfw add pipe 1 ip from any to any**
- **ipfw pipe 1 config bw 30Kbit/s**

dove la prima istruzione crea la pipe numero 1 della quale fornisce al firewall gli elementi per poter identificare i pacchetti (come si é precedentemente riportato). Mentre con la seconda istruzione viene impostata la politica di gestione del flusso di traffico identificato con la prima.

Per ulteriori informazioni vedere [2] e [3] al fine di poter visionare l'albero completo delle possibilitá di configurazione e analisi di sintassi possibili.

[13] A ben considerare anche in questo caso i comandi *pipe* e *queue* potrebbero essere visti come azioni particolari e cioé ordinano che la gestione del pacchetto venga demandata al traffic shaper.

1.3.2 Dettagli implementativi

L'analisi del comando avviene in maniera sequenziale. Man mano che si riconoscono elementi l'esecuzione viene indirizzata verso un determinato flusso operativo.

Per il riconoscimento l'interfaccia si avvale della *match_token*, un ottima e flessibile funzione che, sulla base di un struttura che associa testo ad un enumerato[14], riconosce con molta adattabilitá i vari comandi di grammatica che l'utente inserisce da riga di comando.

Di particolare importanza é il primo parametro che segue il comando, é questo comando che infatti determina il flusso operativo principale dell'interfaccia utente, in particolare sottolineamo alcuni comandi rilevanti per le identificare alcuni comandi base:

- *add*: l'interfaccia utente si aspetta di dover riconoscere una nuova regola da aggiungere alla lista delle regole attive del firewall
- *enable*/*disable*: l'interfaccia utente si aspetta di dover impostare alcuni comportamenti operativi del firewall
- *show*: l'interfaccia utente visualizza l'intera lista di regole presenti in memoria
- *pipe*/*queue*: l'interfaccia utente si aspetta di ricevere parametri relativi alla configurazione del traffic shaper dummynet

Il comando *add* determina l'esecuzione di un'omonima funzione che si preoccupa del riconoscimento e dell'inserizione di una nuova regola. Riprendendo brevemente la struttura a microistruzioni descritta in 1.1.2, ricordiamo che ogni microistruzione prevede una struttura *ipfw_insn* con un opportuno codice operativo, di seguito chiamato *opcode*. La creazione della regola prevede, come visto nel precedente paragrafo, il riconoscimento sequenziale dei parametri secondo il loro ordine di scrittura utilizzando lo spazio come token separatore tra i singoli comandi, cosí come detta lo standard implementativo dei comandi da shell in Unix. Il primo parametro che segue *add* è l'indicazione del protocollo, il riconoscimento avviene attraverso la scansione del file */etc/protocols*, qualora il protocollo citato non sia IP viene creato un comando di filtraggio con l'opcode *O_PROTO* e come parametro aggiuntivo il numero identificativo del protocollo, in sede di riconoscimento il firewall avrá cura di verificare che il protocollo corrente sia conforme alla richiesta della regola. È da notare che in questa indicazione per protocollo si intende sempre il protocollo a livello di trasporto, per questo motivo il firewall cosí come è implementato non prevede il supporto ad IPv6, tranne nel caso in cui il pacchetto in questione non sia incapsulato in un tunnel IPv4.

Dopo l'analisi del protocollo viene analizzato richiesto l'indirizzo sorgente e destinazione che dovrá essere controllato nel pacchetto. Qui sono possibili aggregazioni di indirizzi che dal punto di vista implementativo sono del tutto simili. A seconda della modalitá di inserimento dell'indirizzo, esso verrá tradotto in con un differente opcode

[14] In termini *STL* una tale struttura si chiama *map*, si puó considerarsi come, appunto, una mappatura di funzione biettiva che lega il testo da riconoscere con un enumerato piú semplice da gestire informaticamente.

per facilitare sia la programmazione del firewall sia per ottimizzare l'occupazione di memoria dell'indirizzo stesso. Gli indirizzi possono essere nelle seguenti forme:

- indirizzo singolo e verraá tradotto con un opcode del tipo *O_IP_**[15]
- indirizzo con maschera e verraá tradotto con un opcode del tipo *O_IP_*_MASK*
- lista di indirizzi e verrá tradotto con un opcode del tipo *O_IP_*_SET* ed avrà come parametro il numero di entry inserite
- **me** e verrá tradotto con un opcode del tipo *O_IP_*_ME*
- **any** questo è un caso speciale e viene tradotto lasciando il tutto vuoto.

Successivamente la funzione *add* si aspetta una qualsiasi opzione la cui interpretazione genera un corrispondente opcode. In questo caso l'ordine di digitazione di un comando rispetto ad un altro non ha importanza, in ogni caso la generazione dell'opcode corrispondente avviene secondo l'ordine di digitazione. Sará cura del firewall controllare che tutti gli opcode della regola effettuino il match con il pacchetto desiderato. Una volta completata la creazione della regola in ogni sua parte, questa viene inviata alla parte controllo del firewall attraverso un socket comunicativo. Successivamente viene invocata la visualizzazione della regola stessa, tale visualizzazione è peró dipendente dal firewall nel senso che viene richiesto al firewall di restituire la regola appena inserita. Questo, che a prima vista potrebbe sembrare una perdita di tempo, è un meccanismo molto utile durante sia la stesura delle regole che durante lo sviluppo dei meccanismi di filtraggio, infatti si ottiene un immediato feedback sulla correttezza di quanto si é inserito poiché l'inserimento all'interno della lista delle regole e la successiva richiesta di visualizzazione avviene attraversando il firewall che funge da tester dei dati inviati.

La presenza di un comando *enable* o *disable* determina il conoscimento di una stringa operativa corrispondente ad una variabile *sysctl* e la relativa impostazione secondo le convenzioni di FreeBSD. La gestione di queste opzioni è completamente riferita ai meccanismi standard di gestione del kernel, per cui il riconoscimento e l'attivazione/disattivazione di queste impostazioni non presenta particolari difficoltá.

Il comando *show* attiva, invece, la visualizzazione delle regole del firewall. La visualizzazione viene gestita dal modulo *list*, questo modulo interroga il firewall e si fa restituire la lista di regole dopo di che provvede a visualizzarle analizzando tutti gli opcode presenti e rappresentando l'uscita in modo tale che utilizzando il risultato delle elaborazioni all'interno di un file, e successivamente ordinando all'interfaccia grafica di interpretarlo come file di configurazione si possa ri-ottenere la medesima lista di regole. Questa funzionalitá è estremamente interessante in fase di configurazione della postazione, l'operatore può quindi proccuparsi di testare dinamicamente il set di regole e, una volta raggiunta la sicurezza del funzionamento, salvare opportunamente l'uscita del comando per ri-ottenere la configurazione appena testata.

[15]Con la scritta * vogliamo intendere che il corretto opcode dipenderá dal tipo di indirizzo digitato e cioé se si tratta di un indirizzo di sorgente avremo *O_IP_SRC* mentre se si tratta di un indirizzo di destinazione si avrá *O_IP_DST*. Questa notazione é applicata anche alle successive indicazioni.

La gestione diretta di dummynet la sua attivazione avviene inserendo come prima opzione di uno dei parametri *pipe* o *queue*. Per la sua gestione sono presenti dei comandi standard per la gestione delle pipe, quali cancellazione spostamento ed, ovviamente, aggiunta e configurazione.

Il tipo di configurazione rispecchia la divisione operativa spiegata in 1.2.1. Il riconoscimento dei parametri relativi alla configurazione avviene come tutti i comandi analizzando sequenzialmente i parametri inseriti e riempiendo man mano la struttura *dn_pipe* che al termine della scansione dei parametri verrá inviata alla parte controllo di dummynet che provvederá ad inserirne i dati. In questo caso, a differenza dell'inserzione di una regola di ipfw2, non abbiamo un feedback immediato della correttezza di quanto avvenuto, ma attraverso una semplice visualizzazione della lista delle pipe si ottiene il medesimo risultato.

Capitolo 2

Internet Protocol Versione 6

2.1 Differenze con IPv4

Internet Protocol Versione 6 [9](piú comunemente IPv6) è l'ultima versione del protocollo internet utilizzato in tutto il mondo, esso sta lentamente sostituendo l'attuale protocollo internet versione 4 (IPv4)[26] allo scopo di venire incontro a dei problemi fondamentali che sono emersi negli anni e per far fronte alle nuove esigenze delle nuove applicazioni multimediali esistenti. I problemi principali derivanti da IPv4 sono l'esaurimento degli indirizzi e l'esplosione delle tabelle di instradamento. Con IPv6 si è risolto il primo problema adottando indirizzi su 128 bit [21] anziché su 32 bit, mentre quello dell'esplosione delle tabelle di instradamento è stato affrontato adottando un'architettura di indirizzamento strettamente gerarchica, sofisticate tecniche di rinumerazione e metodologie per la gestione del multihoming. La nuova architettura definisce tre tipi di indirizzi:

- *unicast* [23]: identificatore per una singola interfaccia. Un pacchetto con un indirizzo unicast è inviato all'interfaccia identificata da tale indirizzo;
- *multicast*[18]: identificatore per un insieme di interfacce, tipicamente appartenenti a nodi diversi. Un pacchetto con un indirizzo multicast è inviato a tutte le interfacce con quell'indirizzo;
- *anycast* [14]: identificatore per un insieme di interfacce, tipicamente appartenenti a nodi diversi. Un pacchetto con un indirizzo unicast è inviato a una (in genere la piú vicina, secondo le misure dei protocolli di routing) delle interfacce identificate da tale indirizzo;

  Dal punto di vista della sicurezza IPv6 prevede al suo interno ed in maniera nativa il supporto IPSEC con tutti i vantaggi che se ne ricavano.

2.1.1 Header

Analizziamo ora l'header IPv4 ([26]) e l'header IPv6 ([9]) valutandone le differenze.

Rispetto ad IPv4, il formato dell'intestazione è piú semplice e questo permette migliori prestazioni.

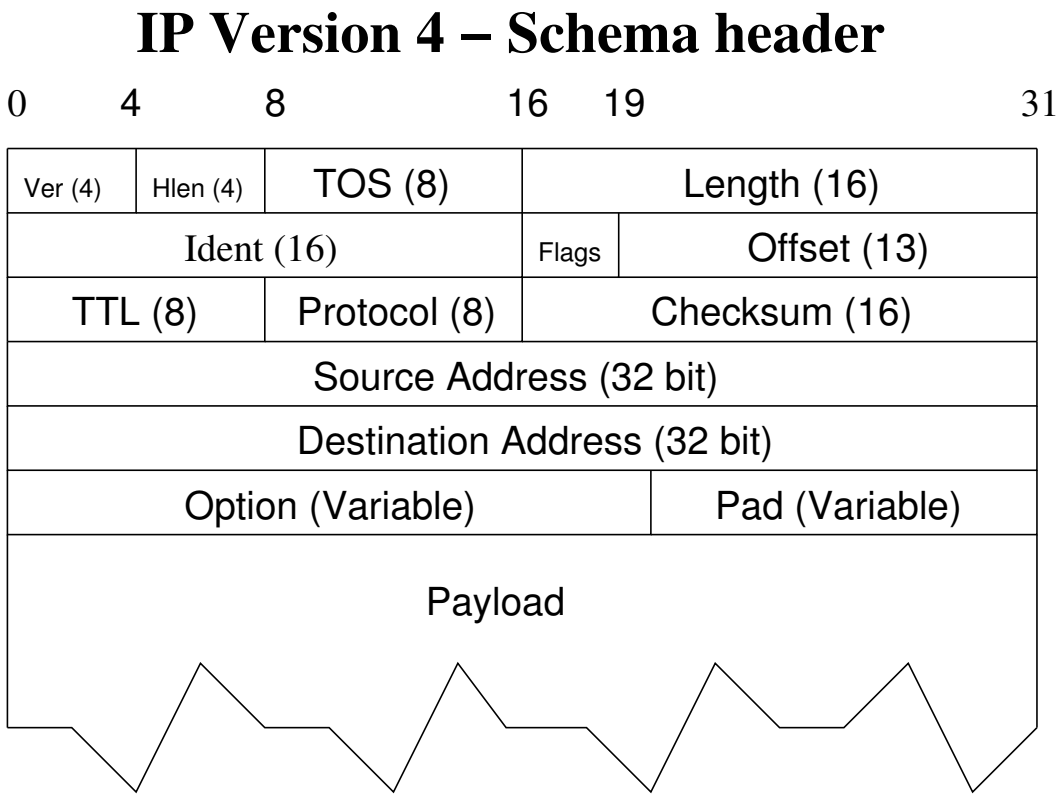

Figura 2.1: Formato header IPv4

Analizziamo i campi piú significativi:

Traffic Class: questo campo è utilizzato dalla sorgente e dai router per identificare i pacchetti che appartengono a una stessa classe di traffico e quindi distinguere tra loro i pacchetti con diversa prioritá.

Flow Label: etichetta un flusso di dati (un parametro di significato molto analogo a quello del protocollo ATM). L'introduzione di questo parametro permette l'individuazione dei dati che richiedono trattamenti particolari, quali per esempio quelli delle applicazioni real time.

Bisogna osservare che rispetto ad IPv4 è stato eliminato il campo *checksum*. La scelta di eliminare il checksum deriva dal fatto che esso è giá calcolato a livello 2 e, dato il tasso di errore delle reti attuali, tale controllo è sufficiente. Si ottengono cosí migliori prestazioni poiché i router non devono piú ricalcolarlo per ogni pacchetto. Eliminando il checksum non si è peró tutelati dagli errori che i router possono commettere nel processare i pacchetti. Tali errori comunque non risultano pericolosi per la rete in quanto causano solo la perdita del pacchetto stesso in seguito alla presenza di campi con valori non validi (es: indirizzi inesistenti).

È stato eliminato il campo *header lenght*, questo perché In IPv4 la lunghezza dell'intestazione è variabile, quindi bisogna specificare sia la lunghezza dell'header IPv4 che la misura totale del pacchetto (total length). In IPv6, invece, l'intestazione ha una dimensione fissa pari a 40 byte quindi è sufficiente indicare solo la lunghezza del campo dati. Il campo payload length è lungo 16 bit, quindi il pacchetto non puó superare i 64 kb. Tale dimensione garantisce ancora buone prestazioni per i router (attesa

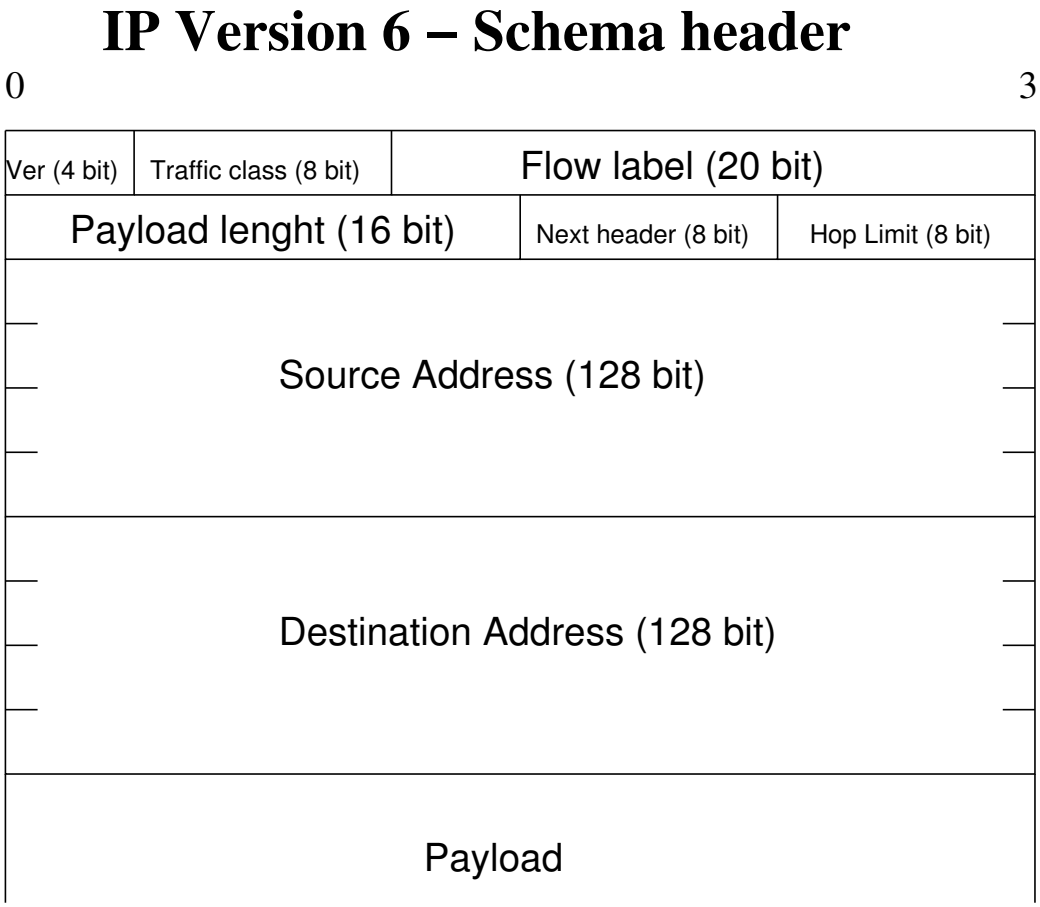

Figura 2.2: Formato header IPv6

in coda limitata, overhead dello 0.06%), ma tale limite è troppo stringente quando a comunicare sono dei supercomputer. Essi sono dotati di memorie enormi e in genere sono collegati tra loro direttamente, quindi sarebbe comodo poter avere pacchetti molto maggiori di 64 kb. Per le loro esigenze è stata pensata l'opzione *jumbogram*[1] in cui il campo payload length è posto a zero e il pacchetto puó superare i limiti imposti.

È stato eliminato il campo *options* rispetto ad IPv4, questo perchè essendo la lunghezza dell'intestazione fissa, si prevede un meccanismo di extension header per l'inserimento delle opzioni. Il tempo di elaborazione dei pacchetti IPv4 aumenta, in quanto ogni router attraversato deve processare tutti gli options prima di inoltrare il pacchetto. Per soddisfare l'esigenza di poter specificare comunque delle opzioni in modo pié efficiente rispetto al passato, si è pensato al meccanismo degli extension header [9]. Per ora ne sono stati specificati sei, ma in futuro se ne potranno aggiungere altri. Gli extension header sono inseriti subito dopo l'header IPv6, in questo modo per i router che non li devono processare essi fanno parte del payload del pacchetto e non sono analizzati. Per questo motivo si ha un miglioramento delle prestazioni di forwarding. Ogni pacchetto puó contenere piú di una intestazione, come mostrato in figura 4; ognuna deve specificare, nel campo next header, il tipo della successiva, for-

[1] Informazioni piú dettagliate [15]

mando quella che viene chiamata ”catena di header”. Nel formare tale catena occorre rispettare il seguente ordine:

- header IPv6;
- hop by hop option header
- destination option header
- routing header
- fragmentation header
- authentication header
- encrypted security payload header
- destination option header
- upper layer header (es. TCP o UDP).

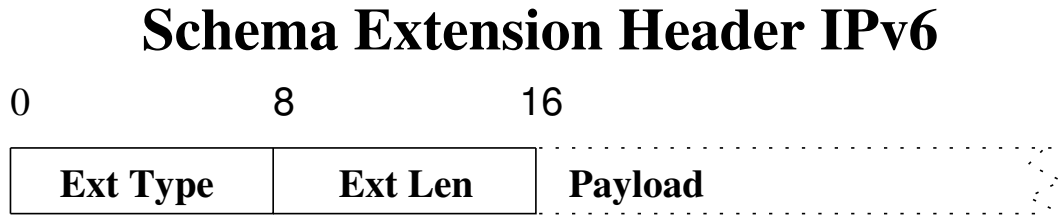

Figura 2.3: Schema extension header

Sono stati eliminati i campi riferiti alla frammentazione rispetto ad IPv4, questo perché mediante il meccanismo degli extension header visto precedentemente, viene gestita anche la frammentazione dei pacchetti e questa rappresenta una differenza fondamentale rispetto IPv4. In questo modo infatti, non solo si ottiene un'intestazione piú semplice, ma essendo in IPv6 la frammentazione gestita agli estremi, i nodi intermedi possono elaborare i pacchetti piú velocemente.Ció che ha reso possibile fare in modo che la frammentazione avvenga agli estremi é la procedura di MTU Discovery [2]che permette alla sorgente di determinare la MTU minima lungo il cammino di instradamento verso la destinazione. La Maximum Transmission Unit (MTU) indica qual è la dimensione massima che puó avere un pacchetto; tale valore è peculiare di ogni tipo di link, ma in ogni caso il valore minimo per IPv6 è stato fissato a 1280 byte. Questo nuovo meccanismo per la gestione della frammentazione causa, nel caso peggiore in cui il link piú piccolo è l'ultimo, lo spreco di tanti pacchetti quanti sono gli hop, ma è piú conveniente che far gestire la frammentazione ai router.

il protocollo IPv6 prevede anche un meccanismo di *tunneling* [21]per la gestione di pacchetti IPv4 e quindi per mantenere la compatibilitá col vecchio protocollo. In

[2]La procedura è illustrata in maniera dettagliata in [26]

questo modo è possibile avere in rete computer IPv6 insieme a computer IPv4, con i relativi vantaggi (attualmente questa cosa si sta sfruttando per testare IPv6 in rete internet). Vediamo come funziona: supponiamo che un router IPv6 riceva un pacchetto IPv4, in tal caso esso viene incapsulato in un nuovo pacchetto IPv6 (tunnel) prima di essere nuovamente instradato, a cui viene assegnato un indirizzo riservato, appunto per far capire all'hop successivo che dentro quel pacchetto è presente un'altro pacchetto ma IPv4.

2.2 Lo stack protocollare IPv6 in FreeBSD

L'implementazione di tutti i protocolli necessari al funzionamento dell'IPv6 in FreeBSD è stato realizzato dal gruppo Kame[3], questo gruppo sviluppa uno stack per IPv6 compatibile con i sistemi *BSD.

Il gruppo di sviluppo fornisce un supporto completo delle funzionalitá standard di IPv6 fornendo, tra le altre cose, anche un firewall basato però sulla vecchia realizzazione del firewall standard di FreeBSD. Gli agganci e i controlli relativi al loro firewall, che si chiama *ip6fw*, sono presenti nei due moduli principali per la gestione dell'I/O del protocollo e cioè: *ip6_input* e *ip6_output*. Nel capitolo relativo all'implementazione del nostro lavoro spiegheremo con maggiore dettaglio come abbiamo realizzato le modifiche per rendere disponibile i nostri meccanismi filtraggio anche sul layer IPv6.

In questa sede non ci addentreremo nella descrizione dettagliata della realizzazione dello stack protocollare IPv6 in FreeBSD, questo esula dalla scopo del nostro lavoro. Ci basta solo citare la fonte di informazioni da cui abbiamo tratto la scarna documentazione al riguardo.

[3]Informazioni dettagliate sul gruppo Kame sono disponibili in [27] e su [28]

Capitolo 3

Realizzazione

L'implementazione è stata fatta andando a modificare i file esistenti dall'albero dei sorgenti di FreeBSD, per consentire il pieno supporto IPv6, in particolare le modifiche riguardano:

- *IPFW2*: modifiche al firewall per permettere l'intercettazione dei pacchetti IPv6 ed implementazione di nuove regole di filtraggio relative al protocollo IPv6.
- *ipfw*: modifiche all'interfaccia utente in maniera tale da potere supportare un nuovo set di comandi per inserire regole nuove relative al protocollo IPv6
- *Dummynet*: modifiche al traffic shaper per consentire la trattazione ed il filtraggio dei pacchetti IPv6
- *ip6_input*, *ip6_output*, *ip6_forward*: modifiche a queste funzioni dello stack protocollare IPv6 per inserire al loro interno le chiamate a Dummynet ed a IPFW2

Di seguito vedremo in maniera dettagliata le modifiche, per realizzarle abbiamo sfruttato in una prima fase il programma di analisi del codice *cscope* tramite il quale abbiamo analizzato il flusso del programma, successivamente tramite l'editor *vim* e i supporti del compilatore *gcc* abbiamo realizzato e testato le modifiche.

3.1 Modifiche ad IPFW2

Ipfw2 supporta il settaggio sia di regole statiche, cioé di regole che una volta settate rimangono tali finché non vengono esplicitamente cancellate, sia di regole dinamiche che possono morire nell'arco del tempo. Le nostre modifiche hanno interessato tutti e due i tipi di regole allo scopo di un pieno supporto firewall per il protocollo IPv6. Analizziamo ora passo passo le modifiche fatte al modulo IPFW2.

3.1.1 Modifiche per il supporto delle regole statiche

La prima modifica che è stata fatta al firewall consiste nel intercettazione del pacchetto IPv6 proveniente dai vari layer (ethernet, ipv6). Il problema quindi era capire dove andare a prendere le informazioni relative al pacchetto. C'è da dire a tale scopo che FreeBSD memorizza le informazioni provenienti dalla rete in una struttura dati fissa definita a livello di kernel che si chiama struttura **mbuf**. Quindi occorreva prelevare il pacchetto da tale struttura e poi trattarlo secondo i nostri scopi. Per quanto riguarda IPFW2 l'operazione di prelevamento e riconoscimento del pacchetto IPv6 viene effettuata all'interno della funzione *ipfw_chk*:

```
/* Identify ipv6 packets and fill up variables. */
if (pktlen >= sizeof(struct ip6_hdr) &&
        (!args->eh ||
        ntohs(args->eh->ether_type)==ETHERTYPE_IPV6) &&
          mtod(m, struct ip *)->ip_v == 6)
```

dove *pktlen* contiene la dimensione dell'header preso dal mbuf e la funzione *mtod(m,struct ip*)* preleva l'header dal mbuf m. In questa sezione si fa distinzione anche del layer di provenienza del pacchetto in quanto se esso proviene dal layer ethernet, m conterrá anche l'intestazione del protocollo ethernet.

Per fare distinzione tra pacchetti IPv4 e pacchetti IPv6 dopo questo controllo viene settata il flag *is_ipv6*, esso verrá utilizzato successivamente per distinguere i comportamenti del programma a seconda del pacchetto che sta analizzando. Una volta acquisito il pacchetto occorrerá andare a ricavare le informazioni relative al protocollo di livello superiore (ICMPv6, TCP, UDP), per verificare l'esistenza di una eventuale regola di filtraggio per questi protocolli. A tale scopo occorrerá fare una scansione di eventuali extension header per ricavare le informazioni di intestazione di questi protocolli (che sono appunto le informazioni che ci servono per un eventuale filtraggio).

```
/* Search extension headers to find upper layer protocols */
    while (ulp == NULL) {
switch (proto) {
case IPPROTO_ICMPV6:
    PULLUP6(hlen, ulp, struct icmp6_hdr);
    args->f_id.flags = ((struct icmp6_hdr *)ulp)->icmp6_type;
```

```
    break;

case IPPROTO_TCP:
    PULLUP6(hlen, ulp, struct tcphdr);
    dst_port = TCP(ulp)->th_dport;
    src_port = TCP(ulp)->th_sport;
    args->f_id.flags = TCP(ulp)->th_flags;
    break;

case IPPROTO_UDP:
    PULLUP6(hlen, ulp, struct udphdr);
    dst_port = UDP(ulp)->uh_dport;
    src_port = UDP(ulp)->uh_sport;
    break;

case IPPROTO_HOPOPTS:
    PULLUP6(hlen, ulp, struct ip6_hbh);
    ext_hd |= EXT_HOPOPTS;
    hlen += sizeof(struct ip6_hbh);
    proto = ((struct ip6_hbh *)ulp)->ip6h_nxt;
    ulp = NULL;
    break;

case IPPROTO_ROUTING:
    PULLUP6(hlen, ulp, struct ip6_rthdr);
    ext_hd |= EXT_ROUTING;
    hlen += sizeof(struct ip6_rthdr);
    proto = ((struct ip6_rthdr *)ulp)->ip6r_nxt;
    ulp = NULL;
    break;

case IPPROTO_FRAGMENT:
    PULLUP6(hlen, ulp, struct ip6_frag);
    ext_hd |= EXT_FRAGMENT;
    hlen += sizeof (struct ip6_frag);
    proto = ((struct ip6_frag *)ulp)->ip6f_nxt;
    offset = 1;
    ulp = NULL; /* XXX is it correct ? */
    break;

case IPPROTO_AH:
case IPPROTO_NONE:
case IPPROTO_ESP:
    PULLUP6(hlen, ulp, struct ip6_ext);
    if (proto == IPPROTO_AH)
ext_hd |= EXT_AH;
```

```
    else if (proto == IPPROTO_ESP)
ext_hd |= EXT_ESP;
    hlen += ((struct ip6_ext *)ulp)->ip6e_len +
    sizeof (struct ip6_ext);
    proto = ((struct ip6_ext *)ulp)->ip6e_nxt;
    ulp = NULL;
    break;

default:
    printf("IPFW2: IPV6 - Unknown Extension Header (%d)\n",
 proto);
    return 0; /* deny */
    break;
} /*switch */
    }
```

C'è da osservare che in caso di presenza di header di frammentazione non sará possibile verificare la presenza di un protocollo superiore, quindi il firewall **non filtrerá** un'eventuale regola TCP,UDP o ICMPv6 relativa ad un pacchetto frammentato. Il firewall prevede inoltre un filtraggio per extension header e durante tale scansione per ogni extension header trovato viene settato un bit di controllo della variabile *ext_hd* che verrá utilizzato successivamente per verificare un'eventuale regola impostata. Infine il puntatore *ulp* conterrá un puntatore all'header del protocollo superiore.

A questo punto è necessario settare alcuni parametri di traffico che serviranno eventualmente a Dummynet

```
    args->f_id.src_ip6 = mtod(m, struct ip6_hdr *)->ip6_src;
    args->f_id.dst_ip6 = mtod(m, struct ip6_hdr *)->ip6_dst;
    args->f_id.src_ip = 0;
    args->f_id.dst_ip = 0;
    args->f_id.flow_id6 =
ntohs(mtod(m, struct ip6_hdr *)->ip6_flow);
```

Vengono salvati oltre agli indirizzi sorgente e destinazione anche il parametro di traffico di IPv6 *ip6_flow*.

Una volta terminate tutte queste operazioni si passa direttamente alle operazioni di filtraggio vero e proprio che consistono nella scansione dei vari opcode relativi alle regole settate per verificare eventuali match di regole e stabilire se accettare o meno il pacchetto.Questa operazione viene fatta settando la variabile *match*.

```
switch (cmd->opcode) {
/*
 * The first set of opcodes compares the packet's
```

```
 * fields with some pattern, setting 'match' if a
 * match is found. At the end of the loop there is
 * logic to deal with F_NOT and F_OR flags associated
 * with the opcode.
 */
.............................

case O_ICMP6TYPE:
match = is_ipv6 && offset == 0 &&
    proto==IPPROTO_ICMPV6 &&
    icmp6type_match(
((struct icmp6_hdr *)ulp)->icmp6_type,
(ipfw_insn_u32 *)cmd);
break;

.................................

case O_IP6_SRC:
match = is_ipv6 &&
IN6_ARE_ADDR_EQUAL(&args->f_id.src_ip6,
    &((ipfw_insn_ip6 *)cmd)->addr6);
break;

case O_IP6_DST:
match = is_ipv6 &&
IN6_ARE_ADDR_EQUAL(&args->f_id.dst_ip6,
    &((ipfw_insn_ip6 *)cmd)->addr6);

case O_IP6_SRC_MASK:
if (is_ipv6) {
    ipfw_insn_ip6 *te = (ipfw_insn_ip6 *)cmd;
    struct in6_addr p = args->f_id.src_ip6;

    APPLY_MASK(&p, &te->mask6);
    match = IN6_ARE_ADDR_EQUAL(&te->addr6, &p);
}
break;

case O_IP6_DST_MASK:
if (is_ipv6) {
    ipfw_insn_ip6 *te = (ipfw_insn_ip6 *)cmd;
    struct in6_addr p = args->f_id.dst_ip6;

    APPLY_MASK(&p, &te->mask6);
    match = IN6_ARE_ADDR_EQUAL(&te->addr6, &p);
}
```

```
break;

case O_IP6_SRC_ME:
match= is_ipv6 && search_ip6_addr_net(&args->f_id.src_ip6);
break;

case O_IP6_DST_ME:
match= is_ipv6 && search_ip6_addr_net(&args->f_id.dst_ip6);
break;

case O_FLOW6ID:
match = is_ipv6 &&
flow6id_match(args->f_id.flow_id6,
(ipfw_insn_u32 *) cmd);
break;

case O_EXT_HDR:
match = is_ipv6 &&
    (ext_hd & ((ipfw_insn *) cmd)->arg1);
break;

case O_IP6:
match = is_ipv6;
break;
```

Valutiamo i vari case dello switch per spiegare in maniera piú esaudiente alcuni particolari.

- case O_ICMP6TYPE: È l'opcode relativo al filtraggio del tipo di pacchetto ICMPv6. In pratica la verifica della regola sul tipo di messaggio ICMPv6 viene fatta dalla funzione *icmp6type_match*

  ```
  static __inline int
  icmp6type_match (int type, ipfw_insn_u32 *cmd)
  {
        return (type <= ICMP6_MAXTYPE &&
        (cmd->d[type/32] & (1<<(type%32)) ) );
  }
  ```

  la funzione riceve come parametri il numero relativo al tipo di messaggio ICMPv6 del pacchetto ed un puntatore alla stringa relativa alla regola da dove si ricava l'eventuale informazione relativa al tipo del messaggio per il filtraggio.

- case O_IP6_SRC, O_IP6_DST: Sono gli opcode relativi al filtraggio per indirizzo, rispettivamente, sorgente e destinazione. In pratica si va a fare semplicemente un match tra l'indirizzo del pacchetto e della regola
- case O_IP6_SRC_MASK, O_IP6_DST_MASK: Sono gli opcode relativi al filtraggio per indirizzo ma relativamente ad una subnet mask. In pratica si procede come descritto per il caso di indirizzo ma prima di effettuare il match si applica la maschera all'indirizzo del pacchetto ricevuto facendo un and logico.
- case O_IP6_SRC_ME, O_IP6_DST_ME: Sono gli opcode relativi al filtraggio da/al host locale. In pratica se un pacchetto viene inviata o proviene dal host locale viene filtrato. Questa operazione viene in pratica fatta dalla funzione *search_ip6_addr_net*

```
static int
search_ip6_addr_net (struct in6_addr * ip6_addr)
{
struct ifnet *mdc;
struct ifaddr *mdc2;
struct in6_ifaddr *fdm;
struct in6_addr copia;

TAILQ_FOREACH(mdc, &ifnet, if_link)
for (mdc2 = mdc->if_addrlist.tqh_first; mdc2;
    mdc2 = mdc2->ifa_list.tqe_next) {
if (!mdc2->ifa_addr)
continue;
if (mdc2->ifa_addr->sa_family == AF_INET6) {
fdm = (struct in6_ifaddr *)mdc2;
copia = fdm->ia_addr.sin6_addr;
/* need for leaving scope_id in the sock_addr */
in6_clearscope(&copia);
if (IN6_ARE_ADDR_EQUAL(ip6_addr, &copia))
return 1;
}
}
return 0;
}
```

La funzione in pratica si ricava l'indirizzo IPv6 dell'host corrente e lo confronta con quello del pacchetto che è stato passato come parametro.

- case O_FLOW6ID: È l'opcode relativo al filtraggio per flow_id. Questo filtraggio è stato introdotto per permettere di filtrare pacchetti provenienti da determinati flussi di traffico indesiderati. In pratica si va a controllare il tipo di flow id attraverso la funzione *flow6id_match* ed eventualmente avviene il filtraggio.

```
static int
flow6id_match( int curr_flow, ipfw_insn_u32 *cmd )
{
int i;
for (i=0; i <= cmd->o.arg1; ++i )
if (curr_flow == cmd->d[i] )
return 1;
return 0;
}
```

- case O_EXT_HDR: È l'opcode relativo al filtraggio per extension header. In pratica si va a controllare se il bit relativo all'extension header in questione è stato settato (il settaggio eventuale viene fatto nella fase di recupero dell'header del protocollo superiore come visto precedentemente) con un semplice and logico.

3.1.2 Modifiche per il supporto delle regole dinamiche

È stato aggiunto il supporto alle regole dinamiche per IPv6, le modifiche fatte si possono riassumere in tre fondamentali:

- Modifica alla funzione *lookup_dyn_rule*
- Modifica alla funzione *lookup_dyn_parent*
- Modifica alla funzione hash per la memorizzazione delle regole dinamiche *hash_packet*

La prima funzione, serve esclusivamente per la ricerca della regola, eventualmente farne il match, ma anche per verificarne la spirazione. Per quanto riguarda la modifica fatta per il supporto IPv6 è stato aggiunto un controllo sugli indirizzi per permettere il riconoscimento di regole per il nuovo protocollo:

```
 if (IS_IP6_FLOW_ID(pkt)) {
if (IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
&(q->id.src_ip6)) &&
    IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
&(q->id.dst_ip6)) &&
    pkt->src_port == q->id.src_port &&
    pkt->dst_port == q->id.dst_port ) {
dir = MATCH_FORWARD;
break;
}
if (IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
&(q->id.dst_ip6)) &&
    IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
&(q->id.src_ip6)) &&
    pkt->src_port == q->id.dst_port &&
    pkt->dst_port == q->id.src_port ) {
dir = MATCH_REVERSE;
break;
}

      } else {.....................
```

La seconda funzione riguarda l'aggiunta di una regola dinamica per IPv6 ed è stata radicalmente modificata:

```
static ipfw_dyn_rule *
lookup_dyn_parent(struct ipfw_flow_id *pkt,
    struct ip_fw *rule)
{
```

```
ipfw_dyn_rule *q;
int i;

if (ipfw_dyn_v) {
int is_v6 = IS_IP6_FLOW_ID(pkt);
i = hash_packet(pkt);
for (q = ipfw_dyn_v[i] ; q != NULL ; q=q->next)
if (q->dyn_type == O_LIMIT_PARENT &&
    rule== q->rule &&
    pkt->proto == q->id.proto &&
    pkt->src_port == q->id.src_port &&
    pkt->dst_port == q->id.dst_port &&
    (
(is_v6 &&
 IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
&(q->id.src_ip6)) &&
 IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
&(q->id.dst_ip6))) ||
(!is_v6 &&
 pkt->src_ip == q->id.src_ip &&
 pkt->dst_ip == q->id.dst_ip)
    )
) {
q->expire = time_second + dyn_short_lifetime;
DEB(printf("ipfw: lookup_dyn_parent found 0x%p\n",q);)
return q;
}
}
return add_dyn_rule(pkt, O_LIMIT_PARENT, rule);
}
```

In pratica prima dell'inserzione si fa anche qui un controllo sull'indirizzo distinguendo il caso IPv6 da quello IPv4.

La terza ed ultima modifica riguarda la funzione hash per l'inserzione dinamica della regola.

```
static __inline int
hash_packet(struct ipfw_flow_id *id)
{
u_int32_t i;

i = IS_IP6_FLOW_ID(id) ? hash_packet6(id):
(id->dst_ip) ^ (id->src_ip) ^
(id->dst_port) ^ (id->src_port);

i &= (curr_dyn_buckets - 1);
```

```
return i;
}
```

Anche qui è bastato fare una semplice distinzione sugli indirizzi: nel caso si trovi un indirizzo IPv6 viene invovata la funzione *hash_packet6*

```
static __inline int
hash_packet6(struct ipfw_flow_id *id)
{
u_int32_t i;
i= (id->dst_ip6.__u6_addr.__u6_addr32[0]) ^
(id->dst_ip6.__u6_addr.__u6_addr32[1]) ^
(id->dst_ip6.__u6_addr.__u6_addr32[2]) ^
(id->dst_ip6.__u6_addr.__u6_addr32[3]) ^
(id->dst_port) ^ (id->src_port) ^ (id->flow_id6);
i &= (curr_dyn_buckets - 1);
return i;
}
```

3.1.3 Altre modifiche

Sono state aggiunti gli opcode relativi ad IPv6 alla funzione *check_ipfw_struct*, la quale si occupa di controllare la validità della regola inserita per il modulo ipfw2.

```
case O_IP6_SRC:
case O_IP6_DST:
    if (cmdlen != F_INSN_SIZE(struct in6_addr)
       + F_INSN_SIZE(ipfw_insn))
          goto bad_size;
    break;

case O_FLOW6ID:
  if (cmdlen != F_INSN_SIZE(ipfw_insn_u32) +
   ((ipfw_insn_u32 *)cmd)->o.arg1)
    goto bad_size;
  break;

case O_IP6_SRC_MASK:
case O_IP6_DST_MASK:
    if ( !(cmdlen & 1) || cmdlen > 127)
    goto bad_size;
    break;
case O_ICMP6TYPE:
   if( cmdlen != F_INSN_SIZE( ipfw_insn_icmp6 ) )
      goto bad_size;
   break;
```

È stato aggiunto un algoritmo per il controllo anti spoofing in maniera del tutto simmetrica a quello esistente per IPv4:

```
static int
verify_rev_path6(struct in6_addr *src, struct ifnet *ifp)
{
static struct route_in6 ro;
struct sockaddr_in6 *dst;

dst = (struct sockaddr_in6 * )&(ro.ro_dst);

if ( !(IN6_ARE_ADDR_EQUAL (src, &dst->sin6_addr) )) {
bzero(dst, sizeof(*dst));
dst->sin6_family = AF_INET6;
dst->sin6_len = sizeof(*dst);
dst->sin6_addr = *src;
rtalloc_ign((struct route *)&ro,
```

```
        RTF_CLONING | RTF_PRCLONING);
}
if ((ro.ro_rt == NULL) || (ifp == NULL) ||
    (ro.ro_rt->rt_ifp->if_index != ifp->if_index))
return 0;
return 1;
}

\newpage
```

3.1.4 Modifiche all'header di IPFW2

Per permettere i cambiamenti appena visti, ma anche parte di quelli relativi agli altri moduli, sono state introdotte delle modifiche dell'header di ipfw2 (ip_fw2.h). Vediamoli in dettaglio:

- Introduzione di nuovi opcode relativi ad IPv6: Sono gli opcode visti precedentemente e sono stati introdotti nella struttura *struct ip_fw_opcode*

```
/*
 * More opcodes.
 */
O_IPSEC,            /* has ipsec history */

O_IP6_SRC,          /* address without mask */
O_IP6_SRC_ME,       /* my addresses */
O_IP6_SRC_MASK,     /* address with the mask */
O_IP6_DST,
O_IP6_DST_ME,
O_IP6_DST_MASK,
O_FLOW6ID,   /* for flow id tag in the ipv6 pkt */
O_ICMP6TYPE, /* icmp6 packet type filtering */
O_EXT_HDR,   /* filtering for ipv6 extension header */
O_IP6,
```

- Definizione di codici relativi ai vari Extension Header: Sono utilizzati per il match delle regole relativi agli extension header

```
/*
 * The extension header are filtered only for
 * presence using a bit vector
 * with a flag for each header.
```

```
 */

#define EXT_FRAGMENT 0x1
#define EXT_HOPOPTS 0x2
#define EXT_ROUTING 0x4
#define EXT_AH 0x8
#define EXT_ESP 0x10
```

- Introduzione della struttura *ipfw_insn_ip6* utilizzata per il match delle regole

```
/* Structure for ipv6 */
typedef struct _ipfw_insn_ip6 {
ipfw_insn o;
struct in6_addr addr6;
struct in6_addr mask6;
} ipfw_insn_ip6;
```

- Introduzione della struttura *ipfw_insn_icmp6* per il supporto del protocollo ICMPv6

```
#define IPFW2_ICMP6_MAXV 7
typedef struct _ipfw_insn_icmp6 {
ipfw_insn o;
uint32_t d[IPFW2_ICMP6_MAXV];
} ipfw_insn_icmp6;
```

L'introduzione di questa struttura per il protocollo ICMPv6 si è resa necessaria vista l'implementazione dello stesso nel nuovo standard. In [22] vengono riportate le differenze implementative, che si traducono con un aumento dei tipi di messaggio ed un maggiore spettro di influenza nella configurazione del protocollo. Per venire in contro alle necessitaá di filtraggio che queste modifiche al vecchio standard potrebbero generare, abbiamo deciso di permettere un filtraggio su ogni tipo di messaggio ICMPv6. Il numero massimo di messaggi è stabilito in [22] ed è riportato nella definizione *ICMP6_MAXTYPE* presente nel file *netinet/icmp6.h*, attualmente questo numero è 203. Al fine di permettere il filtraggio contemporaneo di piú messaggi ICMPv6 nella solita regola, abbiamo scelto un'implementazione a vettore di bit. Visto il numero attuale di messaggi, bastano 7 word da 32 bit per coprire l'intero range di possibilitá.

- Aggiunta della struttura *ip6_dn_args*. Questa struttura serve per salvare dei parametri relativi al funzionamento di dummynet e verrá illustrata successivamente
- Aggiunta alla struttura *ip_fw_flow_id* di due campi relativi agli indirizzi IPv6, un campo relativo al flow_id ed un campo *addr_type* per fare distinzione tra indirizzi IPv4 ed IPv6.

3.2 Modifiche a Dummynet

Dummynet è strettamente legato ad IPFW2, tanto che ad occuparsi di preparare i parametri per tutti e due i moduli è IPFW2 stesso. Esso infatti si occupa di riempire nella apposita struttura dati *ipfw_args*, presente come parametro all'interno della struttura dati *dn_pkt* che spiegheremo successivamente, i campi relativi ad IPv6 utilizzati da Dummynet e si occupa anche di fare il match delle regole per esso, attraverso gli opcode *O_PIPE,O_QUEUE*. Le modifiche fatte all'interno di Dummynet hanno essenzialmente riguardato il meccanismo di I/O con i moduli dello stack protocollare IPv6 e l'implementazione di una nuova funzione hash per la creazione delle pipe e delle queue per indirizzi IPv6. Vediamo ora in dettaglio le modifiche fatte.

3.2.1 Modifica al meccanismo di I/O di Dummynet

Le modifiche fatte per con sentire I/O con il layer IPv6 hanno interessato due funzioni:

- *transmit_event*: Questa funzione viene invocata qualora si voglia inserire un pacchetto in una queue, e viene anche chiamata dallo scheduler periodicamente per inviare e/o ricevere pacchetti con il tasso di ritardo stabilito. Risulta chiaro quindi che per effetturare tali operazioni occorre che la funzione stessa comunichi con i moduli dello stack protocollare corrispondente al pacchetto in esame. Le aggiunte fatte quindi riguardano le chiamate corrispondenti alle funzioni di I/O di IPv6, che sono la *ip6_input* e la *ip6_output*.

```
static void
transmit_event(struct dn_pipe *pipe)
{
    struct dn_pkt *pkt ;

    while ( (pkt = pipe->head) &&
            DN_KEY_LEQ(pkt->output_time, curr_time) ) {
/*
 * first unlink, then call procedures, since ip_input()
       * can invoke ip_output() and viceversa, thus
       * causing nested calls
 */
pipe->head = DN_NEXT(pkt) ;

/*
 * The actual mbuf is preceded by a struct dn_pkt,
 * resembling an mbuf (NOT A REAL one, just a
 * small block of malloc'ed memory) with
 *     m_type = MT_TAG
 *     m_flags = PACKET_TAG_DUMMYNET
 *     dn_m (m_next) = actual mbuf to be processed
```

```
 *             by ip_input/output and some other fields.
 * The block IS FREED HERE because it contains
 * parameters passed to the called routine.
 */
switch (pkt->dn_dir) {
case DN_TO_IP_OUT:
    (void)ip_output((struct mbuf *)pkt, NULL, NULL,
                                    0, NULL, NULL);
    rt_unref (pkt->ro.ro_rt) ;
    break ;

case DN_TO_IP_IN :
    ip_input((struct mbuf *)pkt) ;
    break ;

case DN_TO_IP6_IN:
    ip6_input((struct mbuf *)pkt) ;
    break ;

case DN_TO_IP6_OUT:
    (void)ip6_output((struct mbuf *)pkt, NULL, NULL,
                                     0, NULL, NULL, NULL);
     rt_unref (pkt->ip6opt.ro_or.ro_rt) ;

    break ;
.....................................................
```

Per passare i parametri necessari alle funzioni di I/O si è ricorso ad un piccolo stratagemma: queste due funzioni necessitano come parametro ovviamente *mbuf* ma le nostre informazioni sono memorizzate nella struttura *dn_pkt* che peró al suo interno contiene un mbuf, quindi effettuando un cast di puntatore abbiamo risolto il problema.

- *dummynet_io*: Questa funzione si occupa di inserire nelle pipe ed eventualmente crearle, i pacchetti passati.

```
static int
dummynet_io(struct mbuf *m, int pipe_nr, int dir,
struct ip_fw_args *fwa)
{
    struct dn_pkt *pkt;
    struct dn_flow_set *fs;
    struct dn_pipe *pipe ;
    u_int64_t len = m->m_pkthdr.len ;
```

```
    struct dn_flow_queue *q = NULL ;
    int s = splimp();
    int is_pipe;
#if IPFW2
    ipfw_insn *cmd = ACTION_PTR(fwa->rule);

    if (cmd->opcode == O_LOG)
cmd += F_LEN(cmd);
    is_pipe = (cmd->opcode == O_PIPE);
#else
    is_pipe = (fwa->rule->fw_flg & IP_FW_F_COMMAND)
                == IP_FW_F_PIPE;
#endif

    pipe_nr &= 0xffff ;

    /*
     * This is a dummynet rule, so we expect an O_PIPE or
     * O_QUEUE rule.
     */
    fs = locate_flowset(pipe_nr, fwa->rule);
    if (fs == NULL)
goto dropit ; /* this queue/pipe does not exist! */
    pipe = fs->pipe ;
    if (pipe == NULL) { /* must be a queue, try find a
                                * matching pipe */
for (pipe = all_pipes; pipe &&
        pipe->pipe_nr != fs->parent_nr; pipe = pipe->next)
    ;
if (pipe != NULL)
    fs->pipe = pipe ;
else {
    printf("dummynet: no pipe %d for queue %d,
                    drop pkt\n",
fs->parent_nr, fs->fs_nr);
    goto dropit ;
}
    }
    q = find_queue(fs, &(fwa->f_id));
    if ( q == NULL )
goto dropit ; /* cannot allocate queue */
    /*
     * update statistics, then check reasons to
     * drop pkt
     */
    q->tot_bytes += len ;
```

```
    q->tot_pkts++ ;
    if ( fs->plr && random() < fs->plr )
goto dropit ; /* random pkt drop */
    if ( fs->flags_fs & DN_QSIZE_IS_BYTES) {
     if (q->len_bytes > fs->qsize)
    goto dropit ; /* queue size overflow */
    } else {
if (q->len >= fs->qsize)
    goto dropit ; /* queue count overflow */
    }
    if ( fs->flags_fs & DN_IS_RED &&
         red_drops(fs, q, len) )
goto dropit ;

    /* XXX expensive to zero, see if we can remove it*/
    pkt = (struct dn_pkt *)malloc(sizeof (*pkt),
       M_DUMMYNET, M_NOWAIT|M_ZERO);
    if ( pkt == NULL )
goto dropit ; /* cannot allocate packet header */
    /* ok, i can handle the pkt now... */
    /* build and enqueue packet + parameters */
    pkt->hdr.mh_type = MT_TAG;
    pkt->hdr.mh_flags = PACKET_TAG_DUMMYNET;
    pkt->rule = fwa->rule ;
    DN_NEXT(pkt) = NULL;
    pkt->dn_m = m;
    pkt->dn_dir = dir ;

    pkt->ifp = fwa->oif;
    if (dir == DN_TO_IP_OUT) {
/*
 * We need to copy *ro because for ICMP pkts
       * (and maybe others) the caller passed a
       * pointer into the stack; dst might also be
 * a pointer into *ro so it needs to be updated.
 */
pkt->ro = *(fwa->ro);
if (fwa->ro->ro_rt)
    fwa->ro->ro_rt->rt_refcnt++ ;
if (fwa->dst == (struct sockaddr_in *)
              &fwa->ro->ro_dst) /* dst points into ro */
    fwa->dst =
                (struct sockaddr_in *)&(pkt->ro.ro_dst) ;

pkt->dn_dst = fwa->dst;
pkt->flags = fwa->flags;
```

```
    } else if (dir == DN_TO_IP6_OUT) {
memcpy( &(pkt->ip6opt.ro_or), &(fwa->dummypar.ro_or),
sizeof(fwa->dummypar.ro_or));
if (fwa->dummypar.ro_or.ro_rt)
   fwa->dummypar.ro_or.ro_rt->rt_refcnt++;
        if (fwa->dummypar.dst_or ==
    (struct sockaddr_in6 *) &(fwa->dummypar.ro_or.ro_dst));
         fwa->dummypar.dst_or =
(struct sockaddr_in6 *)&(pkt->ip6opt.ro_or.ro_dst);
pkt->ip6opt.dst_or = fwa->dummypar.dst_or;
pkt->ip6opt.flags_or = fwa->dummypar.flags_or;
    }

    if (q->head == NULL)
q->head = pkt;
    else
DN_NEXT(q->tail) = pkt;
    q->tail = pkt;
    q->len++;
    q->len_bytes += len ;

    /* flow was not idle, we are done */
    if ( q->head != pkt )
goto done;
    /*
     * If we reach this point the flow was
     * previously idle, so we need to schedule
     * it. This involves different actions for
     * fixed-rate or WF2Q queues.
     */
    if (is_pipe) {
/*
 * Fixed-rate queue: just insert into the
       * ready_heap.
 */
dn_key t = 0 ;
if (pipe->bandwidth)
    t = SET_TICKS(pkt, q, pipe);
q->sched_time = curr_time ;
if (t == 0) /* must process it now */
    ready_event( q );
else
    heap_insert(&ready_heap, curr_time + t , q );
    } else {
```

È importante osservare che nel caso la pipe rischiesta sia una **pipe di uscita** allora sará necessario salvare dei parametri relativi al pacchetto in questione, quali la entry nella tabella di routing (ro) l'indirizzo sorgente, l'indirizzo destinazione e l'indirizzo dell'interfaccia di rete (ifp) di provenienza. Infatti Dummynet "intercetta" il pacchetto dallo stack inserendole nelle proprie code, compatibilmente con la politica impostata lo reinserirà al momento opportuno. Se in questo intervallo di tempo i dati venissero cancellati non si saprebbe piú dove instradare il pacchetto. A tale scopo si effettua la copia.

3.2.2 Altre Modifiche

È stata introdotta una nuova funzione hash particolare per i pacchetti IPv6 in maniera del tutto simile come è stato fatto per le regole dinamiche di IPFW2. Tali modifiche hanno interessato la funzione *find_queue*:

```
.....................................
if (is_v6) {
    APPLY_MASK(&id->dst_ip6, &fs->flow_mask.dst_ip6);
    APPLY_MASK(&id->src_ip6, &fs->flow_mask.src_ip6);
    id->flow_id6 &= fs->flow_mask.flow_id6;

    i = ((id->dst_ip6.__u6_addr.__u6_addr32[0]) & 0xffff)^
((id->dst_ip6.__u6_addr.__u6_addr32[1]) & 0xffff)^
((id->dst_ip6.__u6_addr.__u6_addr32[2]) & 0xffff)^
((id->dst_ip6.__u6_addr.__u6_addr32[3]) & 0xffff)^

((id->dst_ip6.__u6_addr.__u6_addr32[0] >> 15) & 0xffff)^
((id->dst_ip6.__u6_addr.__u6_addr32[1] >> 15) & 0xffff)^
((id->dst_ip6.__u6_addr.__u6_addr32[2] >> 15) & 0xffff)^
((id->dst_ip6.__u6_addr.__u6_addr32[3] >> 15) & 0xffff)^

((id->src_ip6.__u6_addr.__u6_addr32[0] << 1) & 0xfffff)^
((id->src_ip6.__u6_addr.__u6_addr32[1] << 1) & 0xfffff)^
((id->src_ip6.__u6_addr.__u6_addr32[2] << 1) & 0xfffff)^
((id->src_ip6.__u6_addr.__u6_addr32[3] << 1) & 0xfffff)^

((id->src_ip6.__u6_addr.__u6_addr32[0] << 16) & 0xffff)^
((id->src_ip6.__u6_addr.__u6_addr32[1] << 16) & 0xffff)^
((id->src_ip6.__u6_addr.__u6_addr32[2] << 16) & 0xffff)^
((id->src_ip6.__u6_addr.__u6_addr32[3] << 16) & 0xffff)^

(id->dst_port << 1) ^ (id->src_port) ^
(id->proto ) ^
(id->flow_id6);
.....................................................
```

3.2.3 Modifiche all'header di Dummynet

Le modifiche fatte all'header di Dummynet (ip_dummynet.h) riguardano essenzialmente l'aggiunta di alcuni campi nella struttura *dn_pkt* riguardanti le informazini relative ad IPv6

```
struct dn_pkt {
    struct m_hdr hdr ;
#define DN_NEXT(x) (struct dn_pkt *)(x)->hdr.mh_nextpkt
#define dn_m hdr.mh_next /* packet to be forwarded */

    struct ip_fw *rule; /* matching rule */
    int dn_dir; /* action when packet comes out. */
#define DN_TO_IP_OUT 1
#define DN_TO_IP_IN 2
#define DN_TO_BDG_FWD 3
#define DN_TO_ETH_DEMUX 4
#define DN_TO_ETH_OUT 5
#define DN_TO_IP6_IN 6
#define DN_TO_IP6_OUT 7

    dn_key output_time; /* when the pkt is due for delivery
*/
    struct ifnet *ifp; /* interface, for ip_output */
    struct sockaddr_in *dn_dst ;
    struct route ro; /* route, for ip_output. MUST COPY */
    int flags ; /* flags, for ip_output (IPv6 ?) */
    struct _ip6dn_args ip6opt; /* XXX ipv6 options */
};
```

Sono stati aggiunti i supporti per direzionare i pacchetti verso lo stack protocollare IPv6, rispettivamente *DN_TO_IP6_IN e DN_TO_IP6_OUT*, ma la modifica fondamentale è l'introduzione del campo *ip6opt*, il quale serve per memorizzare i dati al pacchetto in questione, quali la entry nella tabella di routing (ro) l'indirizzo sorgente, l'indirizzo destinazione e l'indirizzo dell'interfaccia di rete (ifp) di provenienza. Questa operazione risulta necessaria ogni qualvolta sia presente una pipe di uscita, infatti Dummynet "intercetta" il pacchetto dallo stack inserendole nelle proprie code, compatibilmente con la politica impostata lo reinserirà al momento opportuno. Se in questo intervallo di tempo i dati venissero cancellati non si saprebbe piú dove instradare il pacchetto. A tale scopo si effettua la copia. La struttura _ip6dn_args è definita nel file *ip_fw2.h*

```
struct _ip6dn_args {
struct route_in6 ro_or;
```

```
int flags_or;
struct ifnet* ifp_or,origifp_or;
struct sockaddr_in6* dst_or;
};
```

3.3 Modifiche all'interfaccia utente

Al fine di rendere disponibili i nuovi meccanismi di filtraggio, si é, come ovvio, proceduto ad aggiornare anche l'interfaccia di creazione delle regole. Come spiegato in 1.3.1 l'interfaccia prevede un meccanismo di riconsocimento sequenziale, che al primo impatto ha creato qualche problema. Il nodo concettuale fondamentale che caratterizzava la precedente realizzazione era il presupporre a priori che il protocollo base fosse IPv4, questo ha determinato una serie di assunzioni a livello di riconoscimento degli indirizzi e di visualizzazione che hanno reso complesso il lavoro di adattamento. Lo sforzo che ha caratterizzato questo aggiornamento è stato anche finalizzato a mantere la compatibilitá con la precedente implementazione e con l'intero set di regole IPv4. In particolare la realizzazione del supporto alle regole IPv6 ha determinato la creazione *ex-novo* di funzioni specifiche e l'adattamento operativo del codice esistente per poter supportare le nuove caratteristiche.

Dal punto di vista concettuale abbiamo cercato di creare una grammatica di riconoscimento per IPv6 che fosse del tutto simile a quella dell'IPv4, questo per cercare di rendere meno traumatico il cambiamento di stile di inserimento delle regole.

3.3.1 Estensione delle funzionalitá presenti

In questo paragrafo analizzaremo le estensioni alle funzionalitá presenti cercando di motivare scelte effettuate.

Estensione delle strutture dati

In 1.3.1 abbiamo spiegato che il meccanismo di riconoscimento, soprattutto per quanto riguarda le *opzioni*, é ottenuto grazie alla funzione *match_token*. Questa sfrutta le strutture che di seguito abbiamo dovuto estendere per aggiungere il supporto IPv6.

Anzitutto nell'enumerato *enum tokens* destinato a contenere tutti i codici relativi ad una qualsivoglia gestione, abbiamo aggiunto dei codici identificati

```
[...]

TOK_IPV6,
TOK_FLOWID,
TOK_ICMP6TYPES,
TOK_EXT6HDR,
TOK_DSTIP6,
TOK_SRCIP6,

[...]
```

che identificano sostanzialmente le tipologie di controllo che abbiamo considerato all'interno di un pacchetto IPv6.

Procedendo poi nella modifica delle strutture relative al riconoscimento dei vari parametri, abbiamo incontrato la struttura *dummynet_params*, destinata a contenere i parametri relativi alla configurazione di dummynet, a questa abbiamo aggiunto:

```
[...]

{ "flow-id",TOK_FLOWID},
{ "dst-ipv6",TOK_DSTIP6},
{ "dst-ip6",TOK_DSTIP6},
{ "src-ipv6",TOK_SRCIP6},
{ "src-ip6",TOK_SRCIP6},

[...]
```

Nella struttura *rule_options* che viene frutttata per riconoscere le opzioni di filtraggio abbiamo aggiunto le seguenti opzioni:

```
[...]

{ "icmp6type",TOK_ICMP6TYPES },
{ "icmp6types",TOK_ICMP6TYPES },
{ "ext6hdr",TOK_EXT6HDR},
{ "flow-id",TOK_FLOWID},
{ "ipv6",TOK_IPV6},
{ "dst-ipv6",TOK_DSTIP6},
{ "dst-ip6",TOK_DSTIP6},
{ "src-ipv6",TOK_SRCIP6},
{ "src-ip6",TOK_SRCIP6},

[...]
```

Le prime due attivano il riconoscimento del filtraggio per i messaggi *ICMP6*, in ordine di digitazione, segue il riconoscimento del filtraggi per *extension header*, per *flow id*, e per gli indirizzi IPv6 di sorgente e destinazione.

Modifiche alle funzioni operative presenti

Questo tipo di modifiche é stato realizzato in quelle funzioni presenti nel codice originale ma che necessitavano aggiornamenti per supportare le nuove funzioni, in pratica abbiamo agito in due settori: nella visualizzazione e nella parte operativa.

Nelle visualizzazione delle regole, non abbiamo aggiunto la gestione dei nuovi opcode. Nel dettaglio nella funzione *ipfw_show* abbiamo inserito:

```
[...]

case O_IP6_SRC:
case O_IP6_SRC_MASK:
case O_IP6_SRC_ME:
show_prerequisites(&flags, HAVE_PROTO6, 0);
if (!(flags & HAVE_SRCIP))
printf(" from");
if ((cmd->len & F_OR) && !or_block)
```

```
printf(" {");
print_ip6((ipfw_insn_ip6 *)cmd,
(flags & HAVE_OPTIONS) ? " src-ip6" : "");
flags |= HAVE_SRCIP | HAVE_PROTO;
break;

case O_IP6_DST:
case O_IP6_DST_MASK:
case O_IP6_DST_ME:
show_prerequisites(&flags, HAVE_PROTO|HAVE_SRCIP, 0);
if (!(flags & HAVE_DSTIP))
printf(" to");
if ((cmd->len & F_OR) && !or_block)
printf(" {");
print_ip6((ipfw_insn_ip6 *)cmd,
(flags & HAVE_OPTIONS) ? " dst-ip6" : "");
flags |= HAVE_DSTIP;
break;

case O_FLOW6ID:
print_flow6id( (ipfw_insn_u32 *) cmd );
flags |= HAVE_OPTIONS;
break;

[...]

case O_IP6:
printf(" ipv6");
break;

case O_ICMP6TYPE:
print_icmp6types((ipfw_insn_u32 *)cmd);
break;

case O_EXT_HDR:
print_ext6hdr( (ipfw_insn *) cmd );
break;

[...]
```

Dove il primo gruppo aggancia la visualizzazione per i comandi di filtraggio primari, mentre il secondo aggancia la visualizzazione per le opzioni relative ai comandi di filtraggio.

Precedentemente abbiamo avuto cura di inserire gli opportuni comandi nella funzione *show_prerequisites*, e cioé

```
[...]
```

```
#define HAVE_PROTO6 0x0080
[...]
if ( !(*flags & HAVE_PROTO) && (want & HAVE_PROTO6))
printf(" ipv6");
[...]
```

Questo per permettere la corretta visualizzazione delle chiamate ad *ipfw_show*.

Sempre nell'ambito della visualizzazione abbiamo abbiamo reso disponibile la visualizzazione per le regole dinamiche su base IPv6, sfruttando la modifica operativa alla struttura *ipfw_flow_id* viste in 3, all'interno della funzione *show_dyn_ipfw* abbiamo quindi inserito la distinzione in fase di visualizzazione dell'indirizzo:

```
[...]

if (!IS_IP6_FLOW_ID(id)) {
a.s_addr = htonl(d->id.src_ip);
printf(" %s %d", inet_ntoa(a), d->id.src_port);

a.s_addr = htonl(d->id.dst_ip);
printf(" <-> %s %d", inet_ntoa(a), d->id.dst_port);
} else {
char buff[255];

inet_ntop(AF_INET6, &(d->id.src_ip6),
buff, sizeof(buff) );
printf(" %s %d", buff, d->id.src_port);

inet_ntop(AF_INET6, &(d->id.dst_ip6),
buff, sizeof(buff) );
printf(" <-> %s %d", buff, d->id.dst_port);
}

[...]
```

Per terminare la parte sulle visualizzazione, abbiamo variato la visualizzazione delle pipe/queue di dummynet qualora all'interno fossero monitorati piú flussi. La visualizzazione é stata scissa in due: prima vengono visualizzati i flussi IPv4 e subito dopo quelli IPv6. Per realizzare questo nella funzione *list_queue* abbiamo inserito:

```
[...]

/*
 * Do IPv4 stuff
 */

for (l = 0; l < fs->rq_elements; l++)
if (!IS_IP6_FLOW_ID(&(q[l].id))) {
struct in_addr ina;
```

```
struct protoent *pe;

if (!index_print) {
index_print = 1;
printf("\n        mask: 0x%02x 0x%08x/0x%04x -> "
                            "0x%08x/0x%04x\n",
    fs->flow_mask.proto,
    fs->flow_mask.src_ip, fs->flow_mask.src_port,
    fs->flow_mask.dst_ip, fs->flow_mask.dst_port);

printf("    BKT Prot ___Source IP/port____ "
"____Dest. IP/port____ Tot_pkt/bytes Pkt/Byte Drp\n");
}
printf("    %3d ", q[l].hash_slot);
pe = getprotobynumber(q[l].id.proto);
if (pe)
printf("%-4s ", pe->p_name);
else
printf("%4u ", q[l].id.proto);
ina.s_addr = htonl(q[l].id.src_ip);
printf("%15s/%-5d ",
    inet_ntoa(ina), q[l].id.src_port);
ina.s_addr = htonl(q[l].id.dst_ip);
printf("%15s/%-5d ",
    inet_ntoa(ina), q[l].id.dst_port);
printf("%4qu %8qu %2u %4u %3u\n",
     q[l].tot_pkts, q[l].tot_bytes,
     q[l].len, q[l].len_bytes, q[l].drops);
if (verbose)
printf("   S %20qd  F %20qd\n",
     q[l].S, q[l].F);
}

/*
 * Do IPv6 stuff
 */

index_print = 0;
for (l = 0; l < fs->rq_elements; l++)
if (IS_IP6_FLOW_ID(&(q[l].id))) {
struct protoent *pe;

if (!index_print) {
index_print = 1;
printf("\n        mask: proto: 0x%02x, flow_id: 0x%08x,  ",
   fs->flow_mask.proto, fs->flow_mask.flow_id6 );
```

```
inet_ntop(AF_INET6, &(fs->flow_mask.src_ip6),
buff, sizeof(buff) );
printf("%s/0x%04x -> ", buff, fs->flow_mask.src_port);
inet_ntop( AF_INET6, &(fs->flow_mask.dst_ip6),
buff, sizeof(buff) );
printf("%s/0x%04x\n", buff, fs->flow_mask.dst_port);

printf("    BKT ___Prot___ _flow-id_ "
"_____________Source IPv6/port_____________ "
"_____________Dest. IPv6/port_____________ "
"Tot_pkt/bytes Pkt/Byte Drp\n");
}
printf("    %3d ", q[l].hash_slot);
pe = getprotobynumber(q[l].id.proto);
if (pe)
printf("%9s ", pe->p_name);
else
printf("%9u ", q[l].id.proto);
printf("%7d  %39s/%-5d ", q[l].id.flow_id6,
inet_ntop(AF_INET6, &(q[l].id.src_ip6),
buff, sizeof(buff)),
q[l].id.src_port);
printf(" %39s/%-5d ",
inet_ntop(AF_INET6, &(q[l].id.dst_ip6),
buff, sizeof(buff)),
q[l].id.dst_port);
printf(" %4qu %8qu %2u %4u %3u\n",
     q[l].tot_pkts, q[l].tot_bytes,
q[l].len, q[l].len_bytes, q[l].drops);
if (verbose)
printf("   S %20qd  F %20qd\n",
q[l].S, q[l].F);
}

[...]
```

La visualizzazione dei due tipi di flusso, seppur basata sul solito concetto, prevede differenze di lunghezza di indirizzi e campi considerevole, questo porta a sostanziali problemi di allineamento, il tutto però puramente visivivo e non operativo. In questo caso abbiamo optato per una visualizzazione coerente per i due protocolli a scapito della leggibilitá, soprattutto per IPv6, in console a bassa risoluzione.

A questo punto vista la parte di visualizzazione, dal punto di vista operativo le modifiche maggiori sono a carico delle funzioni *config_pipe* e *add*.

Per quanto riguarda la prima abbiamo inserito un'analisi mirata alla gestione del nuovo tipo di indirizzi all'interno della maschera di filtraggio del flusso, e cioé modificando la gestione:

```
[...]

case TOK_MASK:
NEED1("mask needs mask specifier\n");
/*
 * per-flow queue, mask is dst_ip, dst_port,
 * src_ip, src_port, proto measured in bits
 */
par = NULL;

bzero(&p.fs.flow_mask, sizeof(p.fs.flow_mask));
end = NULL;

while (ac >= 1) {
    uint32_t *p32 = NULL;
    uint16_t *p16 = NULL;
    uint32_t *p20 = NULL;
    struct in6_addr *pa6 = NULL;
    uint32_t a; /* the mask */

    tok = match_token(dummynet_params, *av);
    ac--; av++;
    switch(tok) {
    case TOK_ALL:
    /*
     * special case, all bits significant
     */
    p.fs.flow_mask.dst_ip = ~0;
    p.fs.flow_mask.src_ip = ~0;
    p.fs.flow_mask.dst_port = ~0;
    p.fs.flow_mask.src_port = ~0;
    p.fs.flow_mask.proto = ~0;
    n2mask( &(p.fs.flow_mask.dst_ip6), 128);
    n2mask( &(p.fs.flow_mask.src_ip6), 128);
    p.fs.flow_mask.flow_id6 = ~0;
    p.fs.flags_fs |= DN_HAVE_FLOW_MASK;
    goto end_mask;

    case TOK_DSTIP:
    p32 = &p.fs.flow_mask.dst_ip;
    break;

    case TOK_SRCIP:
    p32 = &p.fs.flow_mask.src_ip;
    break;
```

```
    case TOK_DSTIP6:
    pa6 =  &(p.fs.flow_mask.dst_ip6);
    break;

    case TOK_SRCIP6:
    pa6 = &(p.fs.flow_mask.src_ip6);
    break;

    case TOK_FLOWID:
    p20 = &p.fs.flow_mask.flow_id6;
    break;

    case TOK_DSTPORT:
    p16 = &p.fs.flow_mask.dst_port;
    break;

    case TOK_SRCPORT:
    p16 = &p.fs.flow_mask.src_port;
    break;

    case TOK_PROTO:
    break;

    default:
    ac++; av--; /* backtrack */
    goto end_mask;
    }
    if (ac < 1)
    errx(EX_USAGE, "mask: value missing");
    if (*av[0] == '/') { /* mask len */
    a = strtoul(av[0]+1, &end, 0);
    /* convert to a mask for non IPv6 */
    if (pa6 == NULL)
    a = (a == 32) ? ~0 : (1 << a) - 1;
    } else /* explicit mask (non IPv6) */
    a = strtoul(av[0], &end, 0);
    if (p32 != NULL)
    *p32 = a;
    else if (p16 != NULL) {
    if (a > 0xffff)
    errx(EX_DATAERR,
"port mask must be 16 bit");
    *p16 = (uint16_t)a;
    } else if (p20 != NULL) {
    if (a > 0xfffff)
errx(EX_DATAERR,
```

```
    "flow_id mask must be 20 bit");
    *p20 = (uint32_t)a;
    } else if (pa6 != NULL) {
    if (a < 0 || a > 128)
errx(EX_DATAERR,
    "in6addr invalid mask len" );
    else
n2mask(pa6, a);
    } else {
    if (a > 0xff)
    errx(EX_DATAERR,
"proto mask must be 8 bit");
    p.fs.flow_mask.proto = (uint8_t)a;
    }
    if (a != 0)
    p.fs.flags_fs |= DN_HAVE_FLOW_MASK;
    ac--; av++;
} /* end while, config masks */
end_mask:
break;

[...]
```

Con questo abbiamo permesso a dummynet, come visto in 3.1.4, di poter sfruttare gli indirizzi IPv6.

Le modifiche interessano la funzione *add* sono le più importanti, in quanto, come si é spiegato in 1.3.1, questa funzione ha il compito di trasferire al firewall *ipfw2* caricato le nuove regole digitate dall'utente o prelevate dal file di configurazione voluto. Alla luce di questo, é chiaro capire che in questa funzione confluiscono sia le modifiche operative per poter supportare la nuova sintassi sia l'inserimento di un nuovo set di funzioni che permettessero la traduzione agevole di quanto l'utente IPv6 deve poter scrivere. Per come abbiamo suddiviso la trattazione dell'implementazione spiegheremo in breve le aggiunte procedurali e riserveremo al prossimo paragrafo il dettaglio relativo alle singole funzioni.

La prima modifica che si incontra per quanto riguarda la funzione *add* è tanto banale quanto operativamente delicata:

```
[...]

if ( add_proto(cmd, *av, &proto) )

[...]
```

La funzione *add* precedente, come si è accennato ad inizio paragrafo, aveva un grosso presupposto operativo rappresentato dall'assunzione di avere a livello IP il solo protocollo IPv4. Con questa assunzione, gli unici indirizzi sorgente e destinazione possibili potevano essere IPv4, quindi in caso di indicazione di protocollo differente, questa

sarebbe stata da riferirsi al livello superiore, ed in ogni caso il successivo indirizzo sorgente e destinazione sarebbe stato sicuramente IPv4. Con l'introduzione di IPv6, questa assunzione non è piú verificata e quindi si é reso necessario introdurre una doppia gestione per capire se il protocollo di livello IP fosse in versione 4 o 6, questo é realizzato mediante la variabile *proto*.

Il cambiamento si rispecchia nella funzione *add_proto* che riportiamo per comoditá di seguito:

```
[...]

static ipfw_insn *
add_proto(ipfw_insn *cmd, char *av, u_char *proto)
{
struct protoent *pe;
        *proto = IPPROTO_IP;

if (!strncmp(av, "all", strlen(av)))
; /* same as "ip" */
else if ((*proto = atoi(av)) > 0)
; /* all done! */
else if ((pe = getprotobyname(av)) != NULL)
*proto = pe->p_proto;
else if(!strncmp(av, "ipv6", strlen(av)) ||
        !strncmp(av, "ip6", strlen(av)) )
{
  *proto = IPPROTO_IPV6;
  return cmd; /* special case for ipv6 */
}
else
return NULL;
if (*proto != IPPROTO_IP && *proto != IPPROTO_IPV6)
fill_cmd(cmd, O_PROTO, 0, *proto);
return cmd;
}

[...]
```

Questa variabile viene riempita secondo il protocollo riportato in fase di inserimento della regola, variando la gestione il protocollo IPv6 è equiparato al protocollo IP, viceversa la scrittura di un protocollo diverso da questi determina la creazione di un comando di filtragggio specifico rivolto all'analisi della presenza dello specificato tipo di protocollo. Questa funzione informa il chiamante circa l'interpretazione data al comando utente.

Tuttavia la variabile non esaurisce lo spettro di possibilitá operative che si presenta a riga di comando. Infatti un generico utente può preoccuparsi di citare protocolli di livello superiore volendo però mantenersi coerente con il filtraggio di indirizzi IPv6

o IPv4 indifferentemente. A questo scopo è stata variata la gestione dell'inserimento degli indirizzi sorgente e destinazione come parte primaria[1] della regola in:

```
[...]

retval = add_src( cmd, *av, proto );

if( retval ){
ac--; av++;
if (F_LEN(cmd) != 0) { /* ! any */
prev = cmd;
cmd = next_cmd(cmd);
}
} else
errx(EX_USAGE, "bad source address %s", *av);

[...]

retval = add_dst(cmd, *av, proto);

if( retval ){
ac--; av++;
if (F_LEN(cmd) != 0) { /* ! any */
prev = cmd;
cmd = next_cmd(cmd);
}
} else
errx( EX_USAGE, "bad destination address %s", *av);

[...]
```

Questo metodo di creazione del comando é stato inserito *ex-novo* con l'unico scopo di permettere l'inserimento di un comando di filtraggio del tipo **indirizzo sorgente e destinazione** cercando di svincolare la funzione *add* dal discernere se questo debbe essere ti tipo IPv4 o IPv6. Al riguardo verranno aggiunti dettagli nel paragrafo successivo quando analizzeremo le funzioni aggiunte.

Dopo aver reso disponibile l'inserzione di indirizzi sorgente e destinazione, abbiamo pensato all'inserimento di opzioni di filtraggio relative ai possibili campi utili in IPv6. Come spiegato in 1.3.2, i campi che ci sono sembrati piú appetibili per un filtraggio dettagliato sono stati gli *extension header*, il *flowid* e il campi della nuova implentazione del protocollo ICMP.

```
[...]
```

[1]La dizione parte primaria verrá compresa in seguito quando si enunceranno le modifiche alle opzioni che sfruttano le medesime funzioni operative

```
case TOK_ICMP6TYPES:
NEED1("icmptypes requires list of types");
fill_icmp6types((ipfw_insn_icmp6 *)cmd, *av);
av++; ac--;
break;

[...]

case TOK_SRCIP6:
NEED1("missing source IP6");
if (add_srcip6(cmd, *av)) {
ac--; av++;
}
break;

case TOK_DSTIP6:
NEED1("missing destination IP6");
if (add_dstip6(cmd, *av)) {
ac--; av++;
}
break;

[...]

case TOK_IPV6:
fill_cmd(cmd, O_IP6, 0, 0);
ac--; av++;
break;

case TOK_EXT6HDR:
fill_ext6hdr( cmd, *av );
ac--; av++;
break;

case TOK_FLOWID:
if (proto != IPPROTO_IPV6 )
errx( EX_USAGE, "flow-id filter is active only for
                                  "ipv6 protocol\n");
fill_flow6( (ipfw_insn_u32 *) cmd, *av );
      ac--;av++;
break;

[...]
```

In questa parte si notano le modifiche per rendere disponibili i filtraggi suddetti, nonché il supporto delle opzioni per rendere disponibili alle regole dinamiche anche il supporto

per IPv6. Quest'ultima funzionalitá è garantita attraverso gli opcode *O_IPV6* e la possibilitá di aggiungere indirizzi IPv6 anche a livello di opzione.

Aggiunta di funzionalitá

A fronte della necessitá di inserire funzionalitá del tutto nuove rispetto alla precedente gestione considerata, abbiamo proceduto cercando di rimanere il piú possibile vicini alla notazione utilizzata nella versione da aggiornare. In questa ottica abbiamo creato qualche funzione dal cui nome si evince il chiaro collegamento operativo con le corrispondenti funzioni dedicate ad IPv4. In particolare:

```
[...]

static void
print_ip6(ipfw_insn_ip6 *cmd, char const *s)
{
struct hostent *he = NULL;
int len = F_LEN((ipfw_insn *) cmd) - 1;
struct in6_addr *a = &(cmd->addr6);
char trad[255];

printf("%s%s ", cmd->o.len & F_NOT ? " not": "", s);

if (cmd->o.opcode == O_IP6_SRC_ME ||
          cmd->o.opcode == O_IP6_DST_ME) {
printf("me6");
return;
}
if (cmd->o.opcode == O_IP6) {
printf(" ipv6");
return;
}

/*
 * len == 4 indicates a single IP, whereas lists of 1
       * or more addr/mask pairs have len = (2n+1). We
       * convert len to n so we use that to count the
       * number of entries.
 */

for (len = len / 4; len > 0; len -= 2, a += 2) {
    int mb =        /* mask length */
(cmd->o.opcode == O_IP6_SRC ||
   cmd->o.opcode == O_IP6_DST) ?
128 : contigmask((uint8_t *)&(a[1]), 128);

    if (mb == 128 && do_resolv)
```

```
he = gethostbyaddr((char *)a, sizeof(*a), AF_INET6);
    if (he != NULL) /* resolved to name */
printf("%s", he->h_name);
    else if (mb == 0) /* any */
printf("any");
    else {/* numeric IP followed by some kind of mask */
if (inet_ntop(AF_INET6,  a,
    trad, sizeof( trad ) ) == NULL)
    printf("Error ntop in print_ip6\n");
printf("%s",  trad );
if (mb < 0) /* XXX not really legal... */
    printf(":%s",
inet_ntop(AF_INET6, &a[1], trad, sizeof(trad)));
else if (mb < 128)
    printf("/%d", mb);
    }
    if (len > 2)
printf(",");
}
}

[...]
```

Si occupa di visualizzare un indirizzo IPv6 con maschera così come l'originale *print_ip* faceva per IPv4.

```
[...]


static void
fill_icmp6types(ipfw_insn_icmp6 *cmd, char *av)
{
uint8_t type;

cmd->d[0] = 0;
while (*av) {
    if (*av == ',')
av++;
    type = strtoul(av, &av, 0);
    if (*av != ',' && *av != '\0')
errx(EX_DATAERR, "invalid ICMP6 type");
    if (type > ICMP6_MAXTYPE)
errx(EX_DATAERR, "ICMP6 type out of range");
    cmd->d[type / 32] |= ( 1 << (type % 32));
}
cmd->o.opcode = O_ICMP6TYPE;
cmd->o.len |= F_INSN_SIZE(ipfw_insn_icmp6);
```

```
}


static void
print_icmp6types(ipfw_insn_u32 *cmd)
{
int i, j;
char sep= ' ';

printf(" ipv6 icmp6types");
for (i = 0; i < IPFW2_ICMP6_MAXV; i++)
for (j=0; j < 32; ++j) {
if ( (cmd->d[i] & (1 << (j))) == 0)
continue;
printf("%c%d", sep, (i*32 + j));
sep = ',';
}
}

[...]
```

Si occupano della visualizzazione e dell'inserizione dei tipi ICMP, che nell'implementazione IPv6 sono differenti sia in posizione che in numero[2]. A questo proposito, vista la numerositá dei messaggi ICMP rilevanti ai fini del filtraggio introdotti con il protocollo in versione 6, si é deciso di implementare la loro inserzione in maniera numerica e non simbolica. Questa scelta è stata presa per motivi di semplicitá implementativa e per non appesantire troppo la stesura di comandi da interfaccia utente. Ovviamente questo tipo di filtraggio permette la gestione simultanea di piú messaggi contemporanei. Per l'implementazione si é scelto un vettore di bit controllato sulla base del numero di messaggio ICMPv6 che si vuole inserire, in fase di filtraggio il numero verrá considearto conformemente alle indicazioni qui riportate.

```
[...]

static void
print_flow6id( ipfw_insn_u32 *cmd)
{
uint16_t i, limit = cmd->o.arg1;
char sep = ',';

printf(" flow-id ");
for( i=0; i < limit; ++i) {
if (i == limit - 1)
sep = ' ';
printf("%d%c", cmd->d[i], sep);
```

[2] In [10] e [13] sono riportati i dettagli implementativi

```
}
}

[...]

/*
 * fills command for ipv6 flow-id filtering
 * note that the 20 bit flow number is stored in a
 * array of u_int32_t it's supported lists of flow-id,
 * so in the o.arg1 we store how many
 * additional flow-id we want to filter, the basic is 1
 */
void
fill_flow6( ipfw_insn_u32 *cmd, char *av )
{
u_int32_t type;         /* Current flow number */
u_int16_t nflow = 0;    /* Current flow index */
char *s = av;
cmd->d[0] = 0;          /* Initializing the base number*/

while (s) {
     av = strsep( &s, ",") ;
type = strtoul(av, &av, 0);
if (*av != ',' && *av != '\0')
errx(EX_DATAERR, "invalid ipv6 flow number %s", av);
if (type > 0xfffff)
errx(EX_DATAERR, "flow number out of range %s", av);
cmd->d[nflow] |= type;
nflow++;
}
if( nflow > 0 ) {
cmd->o.opcode = O_FLOW6ID;
cmd->o.len |= F_INSN_SIZE(ipfw_insn_u32) + nflow;
cmd->o.arg1 = nflow;
}
else {
errx(EX_DATAERR, "invalid ipv6 flow number %s", av);
}
}

[...]
```

Questa coppia di funzioni si occupa del riempimento e della visualizzazione del comando di filtraggio relativo al campo *flow id* dell'header IPv6. Banalmente essendo un campo da 20 bit, abbiamo previsto un contenitore alla potenza di 2 immediatamente piú grande e cioé 32.

```
[...]

/* structure and define for the extension header in ipv6 */
static struct _s_x ext6hdrcodes[] = {
{ "frag",EXT_FRAGMENT },
{ "hopopt",EXT_HOPOPTS },
{ "route",EXT_ROUTING },
{ "ah",EXT_AH },
{ "esp",EXT_ESP },
{ NULL,0 }
};

/* fills command for the extension header filtering */
int
fill_ext6hdr( ipfw_insn *cmd, char *av)
{
int tok;
char *s = av;

cmd->arg1 = 0;

while(s) {
    av = strsep( &s, ",") ;
    tok = match_token(ext6hdrcodes, av);
    switch (tok) {
    case EXT_FRAGMENT:
cmd->arg1 |= EXT_FRAGMENT;
break;

    case EXT_HOPOPTS:
cmd->arg1 |= EXT_HOPOPTS;
break;

    case EXT_ROUTING:
cmd->arg1 |= EXT_ROUTING;
break;

    case EXT_AH:
cmd->arg1 |= EXT_AH;
break;

    case EXT_ESP:
cmd->arg1 |= EXT_ESP;
break;

    default:
```

```
errx( EX_DATAERR, "invalid option for ipv6 exten
headear" );
break;
    }
}
if (cmd->arg1 == 0 )
    return 0;
cmd->opcode = O_EXT_HDR;
cmd->len |= F_INSN_SIZE( ipfw_insn );
return 1;
}

void
print_ext6hdr( ipfw_insn *cmd )
{
char sep = ' ';

printf(" extension header:");
if (cmd->arg1 & EXT_FRAGMENT ) {
    printf("%cfragmentation", sep);
    sep = ',';
}
if (cmd->arg1 & EXT_HOPOPTS ) {
    printf("%chop options", sep);
    sep = ',';
}
if (cmd->arg1 & EXT_ROUTING ) {
    printf("%crouting options", sep);
    sep = ',';
}
if (cmd->arg1 & EXT_AH ) {
    printf("%cauthentication header", sep);
    sep = ',';
}
if (cmd->arg1 & EXT_ESP ) {
    printf("%cencapsulated security payload", sep);
}
}

[...]
```

Relativamente agli *extension header* abbiamo, anche qui, realizzato una coppia di funzioni per l'inserzione del comando di filtraggio relativo e anche la loro visualizzazione. Visto il numero esiguo di header previsti da [9], abbiamo pensato di utilizzare una notazione simbolica per il loro inserimento e visualizzazione.

Procedendo nella descrizione, abbiamo inserito alcune funzioni di supporto:

```
[...]

/* Try to find ipv6 address by hostname */
static int
lookup_host6 (char *host, struct in6_addr *ip6addr)
{
struct hostent *he;

if (!inet_pton(AF_INET6, host, ip6addr)) {
if ((he = gethostbyname2(host, AF_INET6)) == NULL)
return(-1);
memcpy( ip6addr, he->h_addr_list[0],
                          sizeof(struct in6_addr));
}
return(0);
}

/* n2mask sets n bits of the mask */

static void
n2mask(struct in6_addr *mask, int n)
{
static int minimask[9] = {
    0x00, 0x80, 0xc0, 0xe0, 0xf0, 0xf8, 0xfc, 0xfe, 0xff
};
u_char  *p;
int     i;

memset(mask, 0, sizeof(struct in6_addr));
p = (u_char *) mask;
for (i = 0; i < 16; i++, p++, n -= 8) {
if (n >= 8) {
*p = 0xff;
continue;
}
*p = minimask[n];
break;
}
return;
}

/*
 * fills the addr and mask fields in the instruction as
 * appropriate from av.
 * Update length as appropriate.
 * The following formats are allowed:
```

```
 *     any     matches any IP6. Actually returns an
 *               empty instruction.
 *     me      returns O_IP6_*_ME
 *
 * 03f1::234:123:0342          single IP6 address
 * 03f1::234:123:0342/24      address/mask
 * 03f1::234:123:0342/24,03f1::234:123:0343/5
 *                     List of address
 *
 * Set of address (as in ipv6) not supported because
 * ipv6 address are typically random past the initial
 * prefix.
 *
 * Return 1 on success, 0 on failure.
 */

static int
fill_ip6(ipfw_insn_ip6 *cmd, char *av)
{
int len = 0;
struct in6_addr *d = &(cmd->addr6);
/* Needed for multiple address.
 * Note d[1] points to struct in6_add r mask6 of cmd
 */

cmd->o.len &= ~F_LEN_MASK;      /* zero len */

if (!strncmp(av, "any", strlen(av)))
return 1;

/* Set the data for "me" opt*/
if (!strncmp(av, "me", strlen(av))) {
cmd->o.len |= F_INSN_SIZE(ipfw_insn);
return 1;
}
      /* Set the data for "me" opt*/
if (!strncmp(av, "me6", strlen(av))) {
cmd->o.len |= F_INSN_SIZE(ipfw_insn);
return 1;
}

av = strdup(av);
while (av) {
/*
 * After the address we can have '/' indicating a mask,
 * or ',' indicating another address follows.
```

```
 */

char *p;
int masklen;
char md = '\0';

if ((p = strpbrk( av, "/,")) ) {
md = *p; /* save the separator */
*p = '\0'; /* terminate address string */
p++; /* and skip past it */
}
/* now p points to NULL, mask or next entry */

/* lookup stores address in *d as a side effect */
if (lookup_host6(av, d) != 0) {
/* failed. Free memory and go */
errx(EX_DATAERR, "bad address \"%s\"", av);
}
/* next, look at the mask, if any */
masklen = (md == '/') ? atoi(p) : 128;
if (masklen > 128 || masklen < 0)
errx(EX_DATAERR, "bad width \"%s\'", p);
else
n2mask( &d[1], masklen);

APPLY_MASK( d, &d[1]) /* mask base address with mask */

/* find next separator */

if (md == '/') { /* find separator past the mask */
p = strpbrk(p, ",");
if (p)
p++;
}
av = p;

/* Check this entry */
if (masklen == 0) {
    /*
     * 'any' turns the entire list into a NOP.
     * 'not any' never matches, so it is removed from the
     * list unless it is the only item, in which case we
     * report an error.
     */
    if (cmd->o.len & F_NOT) { /* "not any" never matches */
if (av == NULL && len == 0) /* only this entry */
```

```
errx(EX_DATAERR, "not any never matches");
    }
    /* else do nothing and skip this entry */
    continue;
}

/*
 * A single IP can be stored alone
 */
if (masklen == 128 && av == NULL && len == 0) {
    len = F_INSN_SIZE(struct in6_addr);
    break;
}

/* Update length and pointer to arguments */
len += F_INSN_SIZE(struct in6_addr)*2;
d += 2;
} /* end while */

/* Total lenght of the command, remember that 1 is
       * the size of the base command */
cmd->o.len |= len+1;
free(av);
return 1;
}

[...]
```

La *lookup_host6* effettua la traduzione da **presentation format** a **network format** dell'indirizzo IPv6 specificato, la *n2mask* effettua la traduzione per una maschera da numero a maschera di bit per un indirizzo IPv6, mentre la piú rilevante *fill_ip6* è la funzione cui è delegato il compito di riempire la struttura del comando di filtraggio relativo ad un indirizzo sia esso sorgente o destinazione. Quest'ultima funzione supporta il riempimento per indirizzo singolo, indirizzo + maschera, insiemi di più indirizzi con maschera relativa. Il numero di indirizzi inseribili è formalmente arbitrario ma praticamente limitato a qualche decina.

```
[...]


static ipfw_insn *
add_srcip6(ipfw_insn *cmd, char *av)
{
fill_ip6( (ipfw_insn_ip6 *) cmd, av);
if (F_LEN(cmd) == 0) /* any */
;
if (F_LEN(cmd) == F_INSN_SIZE(ipfw_insn)) /* "me" */
```

```
cmd->opcode = O_IP6_SRC_ME;
else if (F_LEN(cmd) == (F_INSN_SIZE(struct in6_addr)
                + F_INSN_SIZE(ipfw_insn)))
/* single IP, no mask*/
cmd->opcode = O_IP6_SRC;
else /* addr/mask opt */
cmd->opcode = O_IP6_SRC_MASK;
return cmd;
}

static ipfw_insn *
add_dstip6(ipfw_insn *cmd, char *av)
{
fill_ip6((ipfw_insn_ip6 *)cmd, av);
if (F_LEN(cmd) == 0) /* any */
;
if (F_LEN(cmd) == F_INSN_SIZE(ipfw_insn)) /* "me" */
cmd->opcode = O_IP6_DST_ME;
else if (F_LEN(cmd) == (F_INSN_SIZE(struct in6_addr)
                + F_INSN_SIZE(ipfw_insn)))
/* single IP, no mask*/
cmd->opcode = O_IP6_DST;
else /* addr/mask opt */
cmd->opcode = O_IP6_DST_MASK;
return cmd;
}

[...]
```

Queste funzioni, appoggiandosi alla comune *fill_ip6*[3] completano la creazione del comando di filtraggio relativo a seconda che si desideri ottenere un comando che si focalizzi sull'indirizzo sorgente o destinazione.

```
[...]

static ipfw_insn *
add_src(ipfw_insn *cmd, char *av, u_char proto)
{
struct in6_addr a;
        if( proto == IPPROTO_IPV6  ||
           strcmp( av, "me6") == 0 ||
           inet_pton(AF_INET6, av, &a ))
  return add_srcip6(cmd, av);

if (proto == IPPROTO_IP ||
```

[3]Come ovvio che sia visto che l'indirizzo da identificare è il medesimo.

```
          strcmp( av, "me") == 0 ||
          !inet_pton(AF_INET6, av, &a ) )
  return add_srcip(cmd, av);

if( !strcmp( av, "any") )
  return cmd;

return NULL;  /* bad address */
}

static ipfw_insn *
add_dst(ipfw_insn *cmd, char *av, u_char proto)
{
struct in6_addr a;
        if( proto == IPPROTO_IPV6 ||
            strcmp( av, "me6") == 0 ||
             inet_pton(AF_INET6, av, &a ))
  return add_dstip6(cmd, av);

if (proto == IPPROTO_IP ||
          strcmp( av, "me") == 0 ||
          !inet_pton(AF_INET6, av, &a ) )
  return add_dstip(cmd, av);

if( !strcmp( av, "any") )
  return cmd;

return NULL;  /* bad address */
}

[...]
```

Queste funzioni mettono in pratica quanto precedentemente detto a proposito della funzione *add*. Le *add_src* e *add_dst*, infatti, sulla base del protocollo che ottengono e sulla digitazione corrente del comando, prendono una decisione circa l'interpretazione da dare al protocollo attivo e chiamano di conseguenza la funzione deputata alla gestione dell'indirizzo che ci si aspetta al fine di ottenere un comando di filtraggio opportuno.

Il set di modifiche necessarie per l'adeguamento dell'interfaccia con l'utente a questo punto é terminato, infatti, vista la struttura modulare del firewall, non variano le funzioni di variazione delle opzioni operative, quelle di caricamento in memoria della regola, quelle di *fetching* della regola e quelle di gestione.

3.4 Collegamento con il layer IPv6 di FreeBSD 4.x

Le modifiche da fare allo stack protocollare IPv6 per permettere il funzionamento di IPFW2 e DUMMYNET riguardano esclusivamente le funzioni *ip6_input* ed *ip6_output* ed in pratica sono state fatte per i seguenti scopi:

- Intercettare i pacchetti sia in ingresso che in uscita e fare una chiamata al firewall per deciderne il destino.
- Incanalare i pacchetti verso le code di Dummynet, e reinserirli al momento opportuno

Vediamo ora in dettaglio le modifiche fatte.

3.4.1 Inserimento delle chiamate ad IPFW2 e DUMMYNET

Modifiche ad ip6_input

```
/* now check with the firewall ipfw2 */

if (fw_enable && IPFW_LOADED) {
/*
 * If we've been forwarded from the output side, then
 * skip the firewall a second time
 */
args.m = m;
i = ip_fw_chk_ptr(&args);
m = args.m;

if ( (i & IP_FW_PORT_DENY_FLAG) || m == NULL) { /* drop */
if (m)
m_freem(m);
return;
}
ip6 = mtod(m, struct ip6_hdr *); /* just in case m changed */

if (i == 0 && args.next_hop == NULL) /* common case */
goto pass6;
if (DUMMYNET_LOADED &&
     (i & IP_FW_PORT_DYNT_FLAG) != 0) {
/* Send packet to the appropriate pipe */
ip_dn_io_ptr(m, i & 0xffff, DN_TO_IP6_IN, &args);
return;
}
#ifdef IPDIVERT
if (i != 0 && (i & IP_FW_PORT_DYNT_FLAG) == 0) {
/* Divert or tee packet */
```

```
divert_info = i;
ours=1;
}
#endif
if (i == 0 && args.next_hop != NULL)
goto pass6;
/*
 * if we get here, the packet must be dropped
 */
m_freem(m);
return;
}
pass6:
```

Una volta estratto il pacchetto dalla struttura mbuf e fatti gli opportuni controlli viene fatto un salvataggio del mbuf all'interno della struttura *args* di IPFW2 e viene fatta la chiamata al firewall. A seconda del valore di ritorno i il pacchetto verrà accettato o meno. Nel caso in cui anche il traffic shaper DUMMYNET sia attivo, ed esista una regola di traffico per il pacchetto in questione, la variabile *i* conterrà il numero della pipe o della queue di DUMMYNET nella quale andrà a finire il pacchetto e verrà passato come argomento alla chiamata di dummynet. Da osservare che in realtà non viene passato *i* ma *i & 0xffff*, questo per controllare che il numero di pipe non superi il valore limite (le pipe sono limitate). Un'altra cosa da osservare è che subito dopo la chiamata a DUMMYNET c'è un *return*, infatti il pacchetto viene trasferito nelle code di DUMMYNET e ci penserà esso stesso a reinserirlo al momento opportuno dopo averlo opportunamente "segnato" come già analizzato attraverso un TAG. Vediamo cosa succede quando DUMMYNET reinserisce il pacchetto:

```
int    i, hlen, checkif;
u_int32_t divert_info = 0; /* packet divert/tee info */
struct ip_fw_args args;
args.eh = NULL;
args.oif = NULL;
args.rule = NULL;
args.divert_rule = 0; /* divert cookie */
args.next_hop = NULL;

/* Grab info from MT_TAG mbufs prepended to the chain. */
for (; m && m->m_type == MT_TAG; m = m->m_next) {
switch(m->_m_tag_id) {
default:
printf("ip6_input: unrecognised MT_TAG tag %d\n",
    m->_m_tag_id);
break;
```

```
case PACKET_TAG_DUMMYNET:
args.rule = ((struct dn_pkt *)m)->rule;
break;

case PACKET_TAG_DIVERT:
args.divert_rule = (int)m->m_hdr.mh_data & 0xffff;
break;

/* The ipfw2 forwarding is not yet implemented in ipv6

case PACKET_TAG_IPFORWARD:
args.next_hop = (struct sockaddr_in *)m->m_hdr.mh_data;
break;
*/
}
}

KASSERT(m != NULL && (m->m_flags & M_PKTHDR) != 0,
    ("ip6_input: no HDR"));

if (args.rule) { /* dummynet already filtered us */
ip6 = mtod(m, struct ip6_hdr *);
hlen = sizeof (struct ip6_hdr);
goto send_after_dummynet ;
}
```

In pratica appena un pacchetto arriva, ip6_input non può sapere da dove proviene, o meglio, non lo sa ma può verificarlo attraverso una ricerca, e potrebbe anche provenire da DUMMYNET (vedi sopra). A tale scopo viene analizzato tutto mbuf alla ricerca del ”TAG” relativo a DUMMYNET ed in caso affermativo, il pacchetto viene inoltrato subito, visto che era stato già analizzato precedentemente. Un’osservazione abbastanza importante è che le chiamate ad i due moduli sono fatte in maniera tale che il pacchetto non possa ciclare. Infatti se il pacchetto è taggato da DUMMYNET viene segnalata l’ultima regola che ha fatto ”match”. A questo punto, a seconda del valore della variabile di sysctl *net.inet.ip.fw.one_pass*, il firewall deciderà se proseguire la scansione delle regole successive oppure inoltrare il pacchetto, mentre un controllo prima della chiamata a DUMMYNET impedisce un ulteriori cicli.

Modifiche ad ip6_output

```
if (fw_enable && IPFW_LOADED && !args.next_hop) {
/*
 * Check with the firewall IPFW2...
 */
```

```
struct sockaddr_in6 *old = dst;
args.m = m;
args.next_hop = (struct sockaddr_in *) dst;
args.oif = ifp;
off = ip_fw_chk_ptr(&args);
m = args.m;
dst = (struct sockaddr_in6 *) args.next_hop;

/*
 * On return we must do the following:
 * m == NULL    -> drop the pkt (old interface,
 *                          deprecated)
 * (off & IP_FW_PORT_DENY_FLAG)
 *          -> drop the pkt (new interface)
 * 1<=off<= 0xffff              -> DIVERT
 * (off & IP_FW_PORT_DYNT_FLAG)
 *            *            -> send to a DUMMYNET pipe
 * (off & IP_FW_PORT_TEE_FLAG)
 *            *            -> TEE the packet
 * off==0, dst==old             -> accept
 * If some of the above modules are not compiled in,
 * then we should't have to check the corresponding
 * condition (because the ipfw control socket should
 * not accept unsupported rules), but better play safe
 * and drop packets in case of doubt.
 */
if ( (off & IP_FW_PORT_DENY_FLAG) || m == NULL) {
if (m)
m_freem(m);
error = EACCES;
goto done;
}
ip6 = mtod(m, struct ip6_hdr *);
if (off == 0 && dst == old)             /* common case */
goto pass6;
if (DUMMYNET_LOADED &&
   (off & IP_FW_PORT_DYNT_FLAG) != 0) {
/*
 * pass the pkt to dummynet. Need to include
 * pipe number, m, ifp, ro, dst because these are
 * not recomputed in the next pass.
 * All other parameters have been already used and
 * so they are not needed anymore.
 * XXX note: if the ifp or ro entry are deleted
 * while a pkt is in dummynet, we are in trouble!
```

```
 */
args.dummypar.ro_or = *ro;
args.dummypar.flags_or = flags;
args.dummypar.ifp_or = ifp;
args.dummypar.origifp_or = origifp;
args.dummypar.dst_or = *dst;
args.flags = flags;
error = ip_dn_io_ptr(m, off & 0xffff, DN_TO_IP6_OUT,
&args);
goto done;
}
}
pass6:
```

Una volta estratto il pacchetto dalla struttura mbuf e fatti gli opportuni controlli viene fatto un salvataggio del mbuf all'interno della struttura *args* di IPFW2 e viene fatta la chiamata al firewall. A seconda del valore di ritorno i il pacchetto verrà accettato o meno. Nel caso in cui anche il traffic shaper DUMMYNET sia attivo, ed esista una regola di traffico per il pacchetto in questione, la variabile *i* conterrà il numero della pipe o della queue di DUMMYNET nella quale andrà a finire il pacchetto e verrà passato come argomento alla chiamata di dummynet. Da osservare che in realtà non viene passato *i* ma *i & 0xffff*, questo per controllare che il numero di pipe non superi il valore limite (le pipe sono limitate). Un' osservazione molto importante è che prima della chiamata a dummynet vengono salvati dei parametri relativi al pacchetto, quali le interfacce di ricezione e di invio (ifp, orig_ifp, il socket (dst)), i flags del pacchetto e l'entry alla tabella di routing(ro). Questi salvataggi sono **assolutamente necessari!**, infatti quando DUMMYNET reinserirà il pacchetto ripristinerà questi valori altrimenti ip6_output non saprebbe dove instradarlo. Un'altra cosa da osservare è che subito dopo la chiamata a DUMMYNET c'è un *return*, infatti il pacchetto viene trasferito nelle code di DUMMYNET e ci penserà esso stesso a reinserirlo al momento opportuno dopo averlo opportunamente "segnato" come già analizzato attraverso un TAG. Vediamo cosa succede quando DUMMYNET reinserisce il pacchetto:

```
args.eh = NULL;
args.rule = NULL;
args.next_hop = NULL;
args.divert_rule = 0;                    /* divert cookie */

/* Grab info from MT_TAG mbufs prepended to the chain. */
for (; m0 && m0->m_type == MT_TAG; m0 = m0->m_next) {
switch(m0->_m_tag_id) {
default:
printf("ip6_output: unrecognised MT_TAG tag %d\n",
m0->_m_tag_id);
break;
```

```
case PACKET_TAG_DUMMYNET:
/*
 * the packet was already tagged, so part of the
 * processing was already done, and we need to go down.
 * Get parameters from the header.
 */
opt = NULL;
ro = &((struct dn_pkt *)m0)->ip6opt.ro_or;
flags = ((struct dn_pkt *)m0)->ip6opt.flags_or;
                  im6o = NULL;
origifp = ((struct dn_pkt *)m0)->ip6opt.origifp_or;
ifp = ((struct dn_pkt *)m0)->ip6opt.ifp_or;
dst =  &((struct dn_pkt *)m0)->ip6opt.dst_or;
args.rule=((struct dn_pkt *)m0)->rule;
if (args.rule != NULL)
      printf("Collecting parameters\n");
break;

case PACKET_TAG_DIVERT:
args.divert_rule = (int)m0->m_data & 0xffff;
break;

#if 0
/* ipfw2 Forwarding is not yet supported in ipv6 */
case PACKET_TAG_IPFORWARD:
args.next_hop = (struct sockaddr_in *)m0->m_data;
break;
#endif
}
}
m = m0;

KASSERT(!m || (m->m_flags & M_PKTHDR) != 0,
   ("ip6_output: no HDR"));
#ifndef FAST_IPSEC
KASSERT(ro != NULL, ("ip6_output: no route\n"));
#endif

if (args.rule ) {        /* dummynet already saw us */
ip6 = mtod(m, struct ip6_hdr *);
hlen = sizeof (struct ip6_hdr) ;
if (ro->ro_rt)
ia = ifatoia6(ro->ro_rt->rt_ifa);
        bzero(&exthdrs, sizeof(exthdrs));
ro_pmtu = ro;
```

```
goto send_after_dummynet;
}
```

In pratica appena un pacchetto arriva, ip6_output non può sapere da dove proviene, o meglio, non lo sa ma può verificarlo attraverso una ricerca, e potrebbe anche provenire da DUMMYNET (vedi sopra). A tale scopo viene analizzato tutto mbuf alla ricerca del ”TAG” relativo a DUMMYNET ed in caso affermativo, il pacchetto viene inoltrato subito, visto che era stato già analizzato precedentement. Vengono inoltre ripristinati i parametri sopra citati e dopo avere allineato anche i parametri relativi al MTU (maximun transmith unit) viene inviato il pacchetto. Un’osservazione abbastanza importante è che le chiamate ad i due moduli sono fatte in maniera tale che il pacchetto non possa ciclare. Infatti se il pacchetto è taggato da DUMMYNET viene segnalata l’ultima regola che ha fatto ”match”. A questo punto, a seconda del valore della variabile di sysctl *net.inet.ip.fw.one_pass*, il firewall deciderà se proseguire la scansione delle regole successive oppure inoltrare il pacchetto, mentre un controllo prima della chiamata a DUMMYNET impedisce ulteriori cicli.

Capitolo 4

Conclusioni

In conclusione il nostro lavoro ha permesso la creazione di un nuovo Firewall con il pieno supporto ad IPv6, che consente un filtraggio molto fine di pacchetti provenienti da vari stack protocollari, nonchè una ottima funzione di gestione del traffico di rete. L'uso di uno strumento di simile potenza risulta fondamentale in un router di rete, ma anche in un piccolo server, dove per motivi prestazionali IPv6 è largamente utilizzato.

4.0.2 Fase di verifica e test

Per verificare l'attendibilità del lavoro svolto ed eventualmente far emergere bachi nel codice abbiamo provveduto a verificare in maniera meticolosa le varie aggiunte fatte con una serie di prove effettuate presso il laboratorio del Dipartimento di Ingegneria dell'Informazione. Le prove svolte consistono semplicemente nello stabilire delle regole IPv6 tra i vari calcolatori posti in rete tra loro e verificarne il comportamento. C'è da dire che il test dell'interfaccia con l'utente (il parser in pratica) è contestuale agli altri, infatti nel momento in cui viene digitato un comando per stabilire una regola e funziona, allora di conseguenza anche il riconoscimento e l'instradamento di quanto scritto è corretto e coerente, come si è visto in 1.3.1. Vediamo ora in dettaglio le prove.

4.0.3 Test per IPFW2

Il test effettuato ad IPFW2 prevede la verifica fondamentale di due grandi tipologie di regole

- Regole statiche
- Regole dinamiche

Test IPFW2 - regole statiche

La fase di test per quanto riguarda le regole statiche prevede:

- Test sulla regole relativa al filtraggio per indirizzo singolo
- Test sulle regole relative al filtraggio per indirizzi multipli

- Test sulle regole relative al filtraggio per subnet-mask
- Test sulle regole relative al filtraggio da/verso l'interfaccia corrente
- Test sulle regole relative al filtraggio per protocolli superiori(TCP/UDP/ICMPv6)
- Test sulle regole relative al filtraggio per IPv6 flow-id
- Test sulle regole relative al filtraggio per extension-header

Le prove effettuate prevedono l'aggiunta di una regola relativa al test in esame e la verifica della stessa mediante particolari operazioni di rete che prevedono l'utilizzo di essa[1]. C'è da dire che la sola inserzione ed aggiunta della regola con successiva visualizzazione consiste in un vero e proprio test del parser per il riconoscimento e l'invio dei comandi al modulo del firewall, quindi in questo modo la fase di test così fatta copre anche la verifica dell'interfaccia con l'utente. Lo stesso ragionamento è valido per la corretezza delle modifiche fatte ad ip6_input ed ip6_output. Infatti se il firewall filtra bene una regola, per esempio, vuol dire senz'altro che gli stack protocollari passano i pacchetti correttamente, ivi implica un test superato sulla validità delle modifiche effettuate agli stack protocollari.

Test sulla regole relativa al filtraggio per indirizzo singolo

Il test consiste nell'inserire una regola del tipo:

ipfw add deny ipv6 from fe80::250:baff:fe78:5941 to me

Una regola di questo genere impedisce l'ingresso di qualsiasi pacchetto IPv6 proveniente dall'indirizzo fe80::250:baff:fe78:5941 verso l'host locale. Una volta inserita la regola l'interfaccia di visualizzazione mostra il seguente stato:

00100 deny ipv6 from fe80::250:baff:fe78:5941 to me6

Le prove fatte prevedevano il tentativo di ping dall'host di indirizzo fe80::250:baff:fe78:5941, e da altri host estranei alla regola inserita, un tentativo di connessione telnet via TCP, un tentativo di connessione via UDP con il programma traceroute, un tentativo di connessione sicura via ssh. Quest'ultimo era necessario per capire il comportamento del firewall qualora fossimo in presenza di pacchetti criptati SSL. I test sono andati tutti a buon fine, infatti tutti i tentativi di invio di pacchetti dall'host relativo alla regola sono falliti mentre quelli relativi agli altri host sono stati eseguiti correttamente; inoltre il contatore di pacchetti relativo alla regola in questione contava i pacchetti filtrati in maniera coerente.

Test sulle regole relative al filtraggio per indirizzi multipli

Il test consiste nell'inserire una regola del tipo:

ipfw add deny ipv6 from fe80::250:baff:fe78:5941,fe78::250::fe78:3342 to me

[1] I test riportati in questa sede prevedono solo regole per il filtraggio in ingresso. In realtà sono stati fatti anche test equivalenti per le regole in uscita ma essendo totalmente speculari non verranno trattati.

Una regola di questo genere impedisce l'ingresso di qualsiasi pacchetto IPv6 proveniente dagli indirizzi citati, ogni indirizzo deve essere separato da virgola ed il numero di indirizzi previsto è arbitrario. La visualizzazione dello stato della regola è assolutamente equivalente alla precedente con l'unica differenza di avere più indirizzi separati da virgola. Le prove fatte sono essenzialmente identiche alle precedenti con la differenza che i pacchetti filtrati erano relativi a tutti gli hosts presenti nella regola.

Test sulle regole relative al filtraggio per subnet-mask

Il test consiste nell'inserire una regola del tipo

ipfw add deny ipv6 from fe80::250:baff:fe78:5941/80 to me

Una regola di questo genere impedisce l'ingresso all'host corrente di qualsiasi pacchetto IPv6 proveniente dall'indirizzo citato ma relativo ad una subnet-mask in questo caso di 80 bit. Una volta inserita la regola l'interfaccia di visualizzazione mostra il seguente stato:

00100 deny ipv6 from fe80::250:baff:fe78:5941/80 to me6

Le prove effettuate consistono nell'inviare pacchetti tramite ping6,telnet,traceroute6 da hosts tutti con indirizzi IPv6 fe80::250:baff:fe78:5941 ma con subnet-mask diverse. I test sono andati tutti a buon fine, infatti tutti i tentativi di invio di pacchetti dall'host relativo alla regola sono falliti mentre quelli relativi agli altri host sono stati eseguiti correttamente; inoltre il contatore di pacchetti relativo alla regola in questione contava i pacchetti filtrati in maniera coerente.

Test per il filtraggio da/verso l'interfaccia corrente

In pratica è il test relativo all'opcode "me" ed è stato ampiamente illustrato nei test precedenti.

Test per il filtraggio dei protocolli TCP/UDP/ICMPv6

Test relativi a TCP

I test consistono nell'inserire regole del tipo

ipfw add deny tcp from fe80::250:baff:fe78:5941/80 to me

Una regola di questo genere impedisce una qualsiasi connessione tcp il cui pacchetto a livello ip sia IPv6, verso l'host corrente. Una volta inserita la regola l'interfaccia di visualizzazione mostra il seguente stato:

00100 deny tcp from fe80::250:baff:fe78:5941/80 to me6

I test effettuati consistono nel tentativo di connessione tramite telnet all'host sopra citato ma anche nell'invio di pacchetti ICMP6 e UDP tramite rispettivamente ping6 e traceroute. I Test sono andati tutti a buon fine, infatti come ci si aspettava ogni tentativo di connessione TCP è fallito mentre i pacchetti inviati tramite ping6 e traceroute sono stati accettati senza problemi. Inoltre il contatore di pacchetti relativo alla regola imposta contava i pacchetti filtrati in maniera coerente.

Test relativi a UDP

I test consistono nell'inserire regole del tipo

ipfw add deny udp from fe80::250:baff:fe78:5941/80 to me

Una regola di questo genere impedisce una qualsiasi connessione udp il cui pacchetto a livello ip sia IPv6, verso l'host corrente. Una volta inserita la regola l'interfaccia di visualizzazione mostra il seguente stato:

00100 deny udp from fe80::250:baff:fe78:5941/80 to me6

I test effettuati consistono nel tentativo di connessione tramite traceroute all'host sopra citato ma anche nell'invio di pacchetti ICMP6 e TCP tramite rispettivamente ping6 e telnet. I Test sono andati tutti a buon fine, infatti come ci si aspettava ogni tentativo di connessione UDP è fallito mentre i pacchetti inviati tramite ping6 e telnet sono stati accettati senza problemi. Inoltre il contatore di pacchetti relativo alla regola imposta contava i pacchetti filtrati in maniera coerente.

Test relativi a ICMPv6

Il filtraggio dei pacchetti ICMPv6 è molto accurato e prevede non solo il filtraggio di tutti i messaggi provenienti da determinati hosts ma anche per tipo di messaggio. I test consistono nell'inserire regole del tipo

ipfw add deny icmp6 from fe80::350:baff:fe78:5941/80 to me
ipfw add deny icmp6 from fe80::250:baff:fe78:5941/80 to me icmp6types 16,127

La prima regola impedisce il passaggio di un qualsiasi pacchetto ICMPv6 dall'host citato verso l'host corrente; la seconda ivece impedisce il passaggio dei pacchetti ICMPv6 di tipo "16" e ditipo "127". Una volta inserita la regola l'interfaccia di visualizzazione mostra il seguente stato:

00100 deny udp from fe80::250:baff:fe78:5941/80 to me6
00200 deny udp from fe80::250:baff:fe78:5941/80 to me6 icmp6types 16,127

I test effettuati consistono nel tentativo di invio di messaggi ICMPv6 ramite ping6 dagli hosts sopra citati. I Test sono andati tutti a buon fine, infatti come ci si aspettava ogni pacchetto ICMPv6 proveniente da fe80::250:baff:fe78:5941/80 è fallito mentre quelli provenienti da fe80::250:baff:fe78:5941/80 arrivavano regolarmente ad eccezione di quelli di tipo 16 e 127.Inoltre i contatori di pacchetti relativi alle regole imposte contavano i pacchetti filtrati in maniera coerente.

Test sulle regole relative al filtraggio per IPv6 flow-id

I test consistono nel provare regole del tipo:

ipfw add deny ipv6 from any to any flow-id 20,30,50

Tale regola blocca l'accesso sull'host corrente a tutti i datagram IPv6 caratterizzati dai valori di flow-id imposti.I test consistono nell'invio di pacchetti IPv6 caratterizzati da svariati flow-id e verificare se vengono scartati tutti quelli con flow id pari a 50,30,20. Una volta inserita la regola l'interfaccia di visualizzazione mostra il seguente stato:

00100 deny ipv6 from any to any flow-id 20,30,50

I test sono stati fatti con un piccolo programma da noi fatto, visto che non siamo riusciti a trovare in rete un'utilità che venisse incontro a queste esigenze. Il programma da noi fatto in pratica apre un socket UDP con l'host in questione ed invia un pacchetto caratterizzato dal flow-id desiderato. Il test è andato a buon fine.

Test sulle regole relative al filtraggio per extension-header

I test consistono nel provare regole del tipo:

ipfw add deny ipv6 from any to any ext6hdr frag

Tale regola blocca sull'host corrente tutti i pacchetti IPv6 frammentati.È possibile anche inserire altre opzioni separate da virgola come per il flow-id. Una volta inserita la regola l'interfaccia di visualizzazione mostra il seguente stato:

00200 deny ipv6 from any to any extension header: fragmentation

Anche per effettuare questi test siamo stati costretti a creare un'utilità che costruisse un pacchetto IPv6 con l'extension header desiderato, aprisse un socket con l'host in questione e lo inviasse. I test sono andati a buon fine.

Test IPFW2 - regole dinamiche

La fase di test per quanto riguarda le regole dinamiche prevede:

- Test sulle regole relative alla limitazione di connessioni TCP
- Test sulle regole relative alla limitazione di connessioni UDP

Questi tipi di test sono sostanzialmente equivalenti in quanto sono speculari, cambia solo il tipo di connessione. Abbiamo in pratica inserito regole del tipo:

ipfw add allow tcp from fe80::250:baff:fe78:5941 to me setup limit src-addr 4
ipfw add allow udp from fe80::250:baff:fe78:5941 to me setup limit src-addr 4

Le due regole permettono un limite massimo di quattro connessioni, rispettivamente TCP e UDP, all'host corrente.

L'inserimento delle regole provocano una visualizzazione del genere da parte dell'interfaccia utente:

00100 add allow tcp from fe80::250:baff:fe78:5941 to me6 setup limit src-addr 4
00200 add allow udp from fe80::250:baff:fe78:5941 to me6 setup limit src-addr 4

I test fatti consistono nell'effettuare ripetute connessioni di tipo TCP ed UDP attraverso i programmi traceroute e telnet e verificare che superata la quarta connessione le successive venivano rifiutate. I test sono andati a buon fine.

4.0.4 TEST per DUMMYNET

Il test effettuato a DUMMYNET prevede la verifica fondamentale di due grandi tipologie di regole

- Regole relative alle pipe
- Regole relative alle queue

Test DUMMYNET - regole sulle pipe

Questa tipologia di regole è stata testata creando delle pipe con dei ritardi o delle limitazioni di banda, in ingresso ed in uscita, e verificando tramite lo ”show” delle pipe il relativo filtraggio. Per esempio con il comando:

ipfw add pipe 1 ipv6 from me to any
ipfw pipe 1 config bw 30Kbit/s

Crea una pipe dall'host locale (IPv6) verso tutti gli altri limitando la velocità del canale a 30Kbit/s Se per esempio ora proviamo ad effettuare un ping6 in localhost e successivamente scriviamo il comando *ipfw pipe show* visualizzerò:

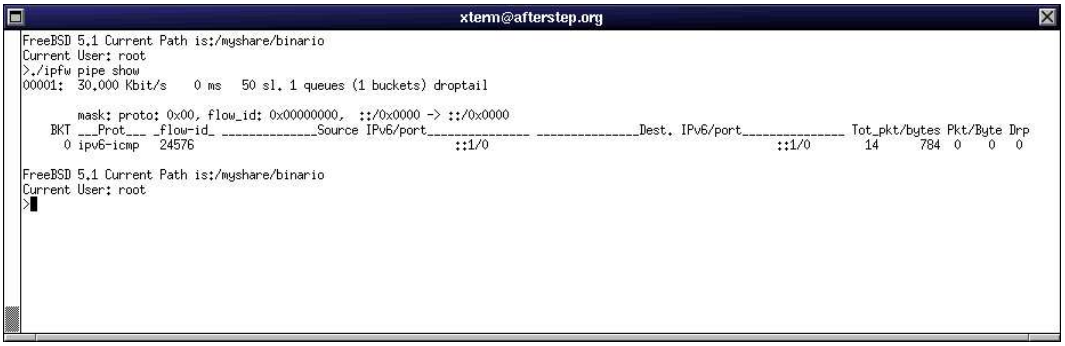

Figura 4.1: Visualizzazione dello stato delle pipe

In pratica questa visualizzazione dimostra il successo del test in quanto la pipe ha filtrato i pacchetti provenienti dal localhost limitando la banda. Per testare in maniera più precisa la limitazione di banda abbiamo effettuato un ping6 e dal valore di delay delle risposte del ping abbiamo verificato il ritardo impostato. I test hanno avuto successo.

Test DUMMYNET - regole sulle queue

Le regole sulle queue sono state verificate creando delle pipe/queue in questo modo:

ipfw add pipe 1 ipv6 from me to any
ipfw pipe 1 config bw 64Kbit/s queue 10Kbytes

Questo comando crea una pipe con limitazione di velocità di 64Kbit/s e di banda bari a 10Kbytes. Effettuiamo ora un ping6 sull'host locale e successivamente lanciamo il comando di visualizzazione della pipe:

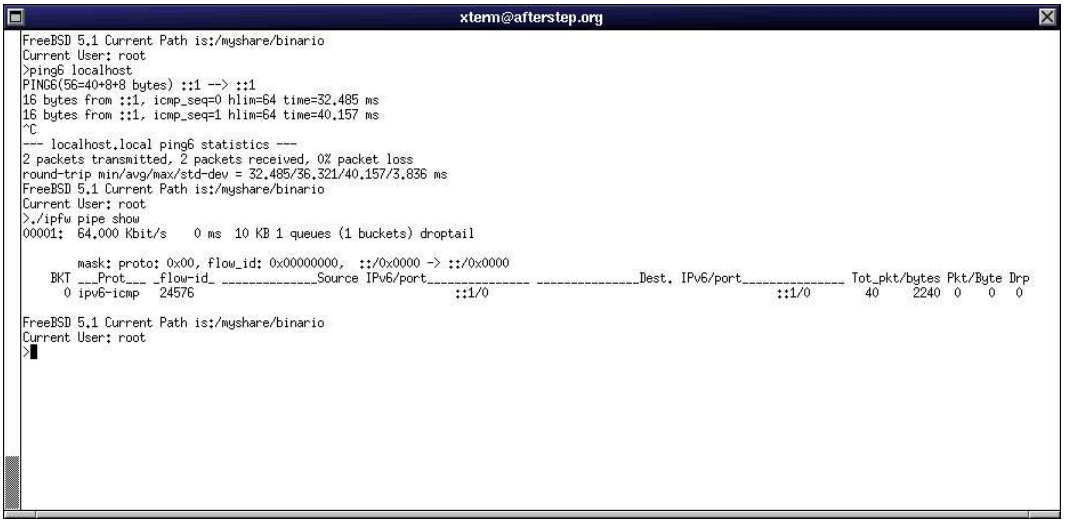

Figura 4.2: Visualizzazione dello stato delle pipe queue

Osserviamo che il traffic shaper ha effettuato egregiamente il suo lavoro, lo si nota sia dalla visualizzazione grafica, sia dal tempo di risposta del ping.

Bibliografia

[1] Paolo Valente - *Sviluppo di un sistema di Fair Queuing in ambiente Unix*, Tesi di laurea, Ottobre 2000

[2] Manuale di sistema su ipfw2: visualizzabile digitando il comando *man ipfw* oppure su internet alla pagina *http://www.FreeBSD.org/cgi/man.cgi?query=ipfw*

[3] Manuale di sistema su dummynet: visualizzabile digitando il comando *man dummynet* oppure su internet alla pagina *http://www.FreeBSD.org/cgi/man.cgi?query=dummynet*.

[4] C. Patridge. *Request for Comment 1809: Using the flow label field in IPv6*, Giugno 1995.

[5] IAB, IESG. *Request for Comment 1881: IPv6 Address Allocation Management*, Dicembre 1995.

[6] Y. Rekhter, T. Li. *Request for Comment 1887: An Architecture for IPv6 Unicast Address Allocation*, Dicembre 1995.

[7] R. Elz. *Request for Comment 1887: A Compact Representation of IPv6 Addresses*, Aprile 1996.

[8] R. Callon, D. Haskin. *Request for Comment 2185: Routing Aspects Of IPv6 Transition*, Settembre 1997.

[9] S. Deering, R. Hinden. *Request for Comment 2460: Internet Protocol, Version 6 (IPv6) Specification*, Dicembre 1998.

[10] A. Conta, S. Deering. *Request for Comment 2463: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*, Dicembre 1998.

[11] M. Crawford. *Request for Comment 2464: Transmission of IPv6 Packets over Ethernet Networks*, Dicembre 1998.

[12] D. Haskin, S. Onishi. *Request for Comment 2465: Management Information Base for IP Version 6: Textual Conventions and General Group*, Dicembre 1998.

[13] D. Haskin, S. Onishi. *Request for Comment 2466: Management Information Base for IP Version 6: ICMPv6 Group*, Dicembre 1998.

[14] D. Johnson, S. Deering. *Request for Comment 2526: Reserved IPv6 Subnet Anycast Addresses*, Marzo 1999.

[15] D. Borman, S. Deering, R. Hinden. *Request for Comment 2675: IPv6 Jumbograms*, Agosto 1999.

[16] R. Hinden, B. Carpenter, L. Masinter. *Request for Comment 2732: Format for Literal IPv6 Addresses in URL's*, Dicembre 1999.

[17] R. Gilligan, E. Nordmark. *Request for Comment 2893: Transition Mechanisms for IPv6 Hosts and Routers*, Agosto 2000.

[18] B. Haberman, D. Thaler. *Request for Comment 3306: Unicast-Prefix-based IPv6 Multicast Addresses*, Agosto 2002.

[19] R. Draves. *Request for Comment 3484: Default Address Selection for Internet Protocol version 6 (IPv6)*, Febbraio 2003.

[20] R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens. *Request for Comment 3493: Basic Socket Interface Extensions for IPv6*, Febbraio 2003.

[21] R. Hinden, S. Deering. *Request for Comment 3513: IPv6 Addressing Architecture*, Aprile 2003.

[22] W. Stevens, M. Thomas, E. Nordmark, T. Jinmei. *Request for Comment 3542: Advanced Sockets Application Program Interface (API) for IPv6*, Maggio 2003.

[23] R. Hinden, S. Deering, E. Nordmark. *Request for Comment 3587: IPv6 Global Unicast Address Format*, Agosto 2003.

[24] B. Wijnen. *Request for Comment 3595: Textual Conventions for IPv6 Flow Label*, Settembre 2003.

[25] J. Rajahalme, A. Conta, B. Carpenter, S. Deering. *Request for Comment 3697: IPv6 Flow Label Specification*, Marzo 2004.

[26] Larry L. Peterson & Bruce S. Davie. *Computer Networks - A System Approach* Second Edition. Morgan Kaufmann Publishers (S. Francisco, California).

[27] The Kame Project - *http://www.kame.org*

[28] Note implementative del progetto Kame visualizzabili presenti nel file */usr/src/share/doc/IPv6/IMPLEMENTATION* disponibile all'interno della distribuzione base di FreeBSD.

Elenco delle figure

Indice